Exploring Ruby and Java Interoperability for Building Converged Web and SIP Applications

Edin Pjanić and Amer Hasanović

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

ABSTRACT

In this paper we present a Ruby infrastructure that can be used for rapid development of Web applications with SIP signaling capabilities. We construct this infrastructure by combining the Java based Cipango SIP/HTTP Servlet Application Server with the Ruby on Rails Web development framework. We provide detailed explanations of the steps required to build this infrastructure and produce a realistic SIP application with an integrated Web interface. The described infrastructure allows Ruby applications to utilize the entire functionality provided by the SIP Servlet API and can be used as a good starting point for the development of Ruby-based domain specific languages for the SIP protocol. We also compare the proposed infrastructure with the existing Ruby frameworks for SIP application development. Furthermore, we present the performance analysis results of the demo application using SIPP, a SIP traffic generator tool.

KEYWORDS
software infrastructure, SIP, converged applications, dynamic languages, interoperability, Ruby, JRuby, Rack, Cipango, Jetty

1 INTRODUCTION

In the nineties, software development was mostly focused on desktop applications utilizing tools, libraries and paradigms built around Java and C++, the two most dominant programming languages of that time. During the last decade, this focus started shifting towards Web applications, agile practices and dynamic programming languages [1][2][3].

The Ruby programming language [4] received a lot of attention in the software industry, primarily because of the flexible Web frameworks, such as Ruby on Rails [5] and Sinatra [6], as well as Ruby’s support for the development of domain specific languages (DSL) [7][8].

With the recent rise of social Web applications, there is an increased interest to bring other services to the Web domain, real-time voice and video in particular. Based on the HTTP model, the Session Initiation Protocol (SIP) [9] was developed to support these services. The telecommunications industry has accepted the SIP protocol, which is now widely implemented and used in telecom application servers.

While Ruby has excellent support for HTTP, due to the Web frameworks, it lacks the adequate support for SIP. On the other hand, Java has well developed APIs, libraries and tools for the SIP protocol, such as the SIP Servlets [10] and JAIN SLEE [11]. Therefore, it would be beneficial to combine Ruby’s proven agility [3] and Java robustness in a single infrastructure for rapid development of
SIP applications. The main concepts of this integration were discussed in [12].

In this paper, we demonstrate the steps required to build the infrastructure which allows hosting of Web applications with real-time VoIP capabilities entirely in Ruby. We reuse well-known and tested open-source components as the basis for this infrastructure. More specifically, we glue together the code provided by the Ruby on Rails Web framework and a Java based Cipango SIP/HTTP Servlet Application Server [13]. We realize this concept with JRuby [14], a Java implementation of the Ruby language. We also provide a detailed explanation of the techniques we use for interoperation between JRuby and Java. Finally, we compare our approach with similar Ruby infrastructures for SIP application development.

The paper is organized as follows. In Section 2 we shortly outline the related infrastructures and compare them with our approach. In Section 3 we focus on interoperability between JRuby and Java, which is essential for utilizing Java libraries from within the Ruby code. In Section 4 we discuss the Cipango SIP/HTTP application server architecture and identify the key classes that can be used to embed the server engine inside a Ruby application. We provide an in-depth procedure to construct the proposed infrastructure by building an example Web application with SIP capabilities in Section 5. In Section 6, we analyze the performance of the demo application. Finally, in Section 7 we provide a conclusion and map a course for the future work.

2 RELATED WORK

At the time of writing this paper, we are aware of the two Ruby infrastructures for SIP application development. In this section we give a brief overview of these frameworks and compare them to our approach.

2.1 TorqueBox

TorqueBox [15] is a Ruby application platform that integrates JRuby Web applications based on Rack [16] frameworks, such as Ruby on Rails, with the JBoss Application Server [17], utilizing the features provided by JBoss. Currently, TorqueBox provides the SIP support in the beta versions.

Building converged applications with TorqueBox is similar to our approach. The main differences are in the application deployment and integration with the SIP application server. This is a direct result of the fact that the TorqueBox uses the JBoss while in our approach we utilize the Cipango application server. Additionally, the TorqueBox application deployer does not support all the features provided by the SIP servlet API from within Ruby code. For instance, listener objects are created and added to the SIP application context during the deployment of the application. The classes for these objects are defined in the sip.xml deployment descriptor file and cannot be Ruby classes.

Application deployment in our approach is completely different. Instead of deploying an application to the server, we incorporate the server inside the Ruby application. This means that we can utilize the full stack of the SIP Servlet
API and control all the features provided by the Cipango application server, including the creation and specification of listener objects, as shown in the demo application. Furthermore, this gives us a possibility to develop and integrate Ruby DSLs in order to speed up and simplify VoIP applications development.

Moreover, the lightweight infrastructure of the Cipango engine allows faster application startup times, which is very important in short development iteration cycles.

2.2 Sipatra

Sipatra [18] is a Ruby DSL for SIP Servlets and targets SIP Servlet 1.1 compatible application servers. Its syntax is much like the syntax used in Sinatra Web DSL, but adapted to support the specific features of the SIP protocol.

Sipatra SIP message handlers are defined as blocks of code that are called when certain conditions are satisfied. The conditions are specified as regular expressions that must match the message’s SIP request method, response code, URI or headers in order for the handler to be invoked. There can be more than one handler for the same SIP message type. The handlers are evaluated in the order they appear in the code and the first handler that matches the condition is executed. An example of the syntax is shown in the Figure 1 listing that is self explanatory.

Similar to our approach, Sipatra uses the Cipango application server. However, the application deployment procedure is similar to the one used in the TorqueBox project. Hence, it suffers from the same limitations discussed in Section 2.1.

```ruby
invite /sip:.@sipatra.org/ do
  ...
end
invite /sip:.@otherdomain.org/ do
  ...
end
invite do
  ...
end
response :INVITE, 200 do
  ...
end
```

Figure 1. Example of the Sipatra syntax.

3 JRUBY AND JAVA INTEROPERABILITY

Matz’s Ruby Interpreter or Ruby MRI is the reference implementation of Ruby, written in C by Yukihiro Matsumoto. However, this is not the only implementation of Ruby. One especially successful open source implementation, called JRuby, is written for the Java Virtual Machine. While being fully compatible with Ruby MRI, it offers certain advantages. First, since it is implemented on top of the Java Virtual Machine (JVM), JRuby threads are mapped to the kernel threads, and Unicode strings are automatically supported in JRuby. Second, JRuby can seamlessly interoperate with Java. Meaning, that we can use Java objects as normal Ruby objects, and vice versa. Hence, we can exploit the wealth of Java libraries, using the power of Ruby’s flexible syntax. To demonstrate the interoperability between JRuby and Java, we use the Java listing shown in Figure 2, in which we define the Foo Java class.

We compile the class and put it in the example.jar library. Now, we can use the Foo class in JRuby. To
```java
package org.test;
public class Foo {
    private int var = 0;
    public void setVar(int v) {
        var = v;
    }
    public int getVar() {
        return var;
    }
    public void method1() {
        System.out.println("In Java \+var");
    }
    public static void method2(Foo obj) {
        obj.method1();
    }
}
```

Figure 2. The example Java class Foo, used to demonstrate the interoperability.

demonstrate this, we write the Ruby program 'test_program.rb', shown in Figure 3.

Since the native Java classes in JRuby are wrapped inside the java module, we include this module in the first line of the script. To access the Foo class, we require example.jar as a Ruby module, and include the class from it's package in lines 2 and 3. Next, we define a Ruby class, called Bar, by subclassing the Foo class. In the Bar class code, we override method1 to print a Ruby message.

We create the two Ruby objects, named java_object and ruby_object, in lines 9 and 10. In line 11, it looks like we are directly setting the var field of ruby_object. This however is not allowed, since we don’t have access to the private fields of the base class. JRuby instead, calls the provided Java setter method setVar, inherited from Foo, and passes value 10 to the method. Similarly, JRuby calls the getter method getVar when we try to read from var in line 6. Finally, we send the two objects to the static Java method of the Foo class, in lines 12 and 13.

```ruby
[1] require 'java'
[2] require 'example.jar'
[3] include_class 'org.test.Foo'
[4] class Bar < Foo
[5]   def method1
[6]     puts "In Ruby \+var.to_s"
[7]   end
[8] end
[9] java_object = Foo.new
[10] ruby_object = Bar.new
[12] Foo::method2 java_object
[13] Foo::method2 ruby_object
```

Figure 3. The code of the 'test_program.rb' script.

When we execute the program, it prints the following output:

```
$ jruby test_program.rb
In Java 0
In Ruby 10
```

We can see that during the second call, when we send ruby_object to the Java world, the Ruby version of method1 gets called inside the code of the Java class.

We use the described interoperability features extensively to embed, configure and manipulate a Java SIP application server inside the code of a converged Web and SIP JRuby application.

4 CIPANGO SIP APPLICATION SERVER

In the telecom Next Generation Networks paradigm, SIP is one of the core protocols used in the Service Delivery Platforms. These platforms can be built around Java Enterprise Edition (EE) [19] servers, extended to support SIP Servlet specification [10]. Using these extensions, applications that support SIP services can be built in Java with SIP servlets, in a similar way Web applications are built with HTTP
servlets. Applications are deployed to Java EE servers in the war archives. The server unpacks the archive, initializes the application and invokes it when an appropriate HTTP or SIP message is received, or when a certain event occurs. Technique similar to this is used in the open source TorqueBox project [15] to deploy JRuby applications to the Mobicents SIP application server [20]. However, with this approach it can be difficult to access from JRuby some features provided by the application server. For instance, creating and controlling the SIP servlet timers.

In this paper, we use another approach. The open source Cipango SIP/HTTP Servlet Application Server, built on top of the Jetty Web Server [21], supports an interesting option of embedding. Using this option, instead of deploying an application to the server, we incorporate the server engine in our application and control it completely.

The Cipango engine consists of the core classes that are shown in Figure 4. To embed the engine, we need to create the following Java objects in a JRuby application:

- one Server instance,
- a number of connector instances that specify the sockets used for communication,
- at least one SipAppContext that represents a context in which we add SIP and HTTP servlets, and finally,
- a number of SipServlet and HttpServlet instances that we use inside the context to handle the specific HTTP and SIP requests and/or responses.

Figure 4. Core Cipango classes

5 THE DEMO APPLICATION

When a SIP VoIP client starts up, it sends a SIP REGISTER request to the SIP registrar server. The request has a dual purpose. It is used to authenticate the user, and provide the server with the current IP location of the user. The user’s current IP address is then saved in a database. Association between the user’s username, or his SIP URI, and the IP address of the machine where the user’s SIP client is running is called binding. Proxy servers use bindings in order to forward requests to SIP clients.

SIP proxy servers have several roles in the SIP infrastructure, such as checking validity of the SIP requests, checking the user’s credentials for certain requests, routing the requests to other proxies or to final recipients. If the proxy is responsible for the final recipient’s domain, it checks the location database for his current IP address and proxies the request to that location in order to deliver the request to the destination.

In this paper, we build a simple registrar and proxy application, using only Ruby. Besides processing the SIP REGISTER and INVITE messages, the application provides a simple Web interface for administration purposes.
We first prepare the Web interface, by using the Rails framework. Then, we create the application in which we embed the Cipango SIP engine. Inside the application code, we configure the Cipango engine to host the Rails generated Web interface, and to handle REGISTER and INVITE SIP messages using the SIP servlet defined in the application.

5.1 Building the Database and Web Interface

We start building the application by downloading the latest JRuby distribution archive from the JRuby Web site [22]. After uncompressing the archive, we add the bin directory of the JRuby distribution to the system path, since we use the command line interface to build the application.

Next, using the gem infrastructure, we install the Rails framework and the SQLite3 database adapter.

```
$ jruby -S gem install rails \
   activerecord-jdbcsqlite3-adapter \
   --no-ri --no-rdoc
```

The previous command installed the latest Rails framework to the JRuby distribution, but without the default documentation.

To build the Web interface from scratch we use the Rails code generators, which we invoke from the command line. For the deeper understanding of these commands, we direct the reader to [23] and [24].

We first build the skeleton Rails project called registrar by issuing the following command:

```
$ jruby -S rails new registrar \
   --template http://jruby.org
```

Based on the template for JRuby Rails applications, this created the registrar project directory, together with the subdirectories that hold the default Rails project infrastructure.

Next, in the project’s directory, in order to construct the Web interface for user administration, we execute the generate scaffold command:

```
$ jruby -S rails generate scaffold \
   sip_user user_name:string \
   first_name:string last_name:string
```

As described in [23], Rails is based on the MVC design pattern. The previous command created the model, view and controller classes for the sip_user resource with the following three fields of the string type: user_name, first_name and last_name.

Now we need to create an additional class that models the SIP registration bindings. The model class has to have a field for the current location of the user and a reference to the user which the binding belongs to. Hence, we create the registration model with a field location and a reference to the sip_user model by issuing the following generate model command:

```
$ jruby -S rails generate model \
   registration location:string \
   sip_user:references
```

The previous generators create SipUser and Resource classes. The Resource class is associated with the SipUser class using the ActiveRecord’s belongs_to association. This is equivalent to a foreign key relationship in relational databases. Now, we need to create the other side of the association by changing the generated SipUser class definition,
saved in the file `app/models/sip_user.rb`
to read:

```ruby
class SipUser < ActiveRecord::Base
  has_one :registration,
    :dependent => :destroy
end
```

This definition means that instance of
the SipUser class can have only one
instance of the Registration class.
Additionally, deletion of the SipUser
instance will initiate deletion of the
related Registration instance
automatically. The association between
those two classes is depicted graphically
in Figure 5.

![Figure 5. Class diagram of the models.](image)

Before we can access the resources
through the Web interface for CRUD
(create, read, update and delete)
operations, we need to build the database
to hold the resource data. The previous
generate commands have also
generated the migration scripts, which
we use to create the SQLite database with
the appropriate schema that describes
the resources. To start the migration
process and generate the production
database, we execute the following Rake
db:migrate command.

```
$ jruby -S rake db:migrate \
   RAILS_ENV=production
```

We are done building the database
and Web interface, which is shown in
Figure 6. Now, we need to add the
required Java libraries to the registrar
project in order to provide the SIP
protocol support for our application.

5.2 Adding the Java Libraries

We download the Cipango application
server distribution, version 1.0.0, from
its Web site [13]. After we uncompress
the Cipango distribution, from the
distribution’s lib directory, we need the
following jar files: cipango, jetty,
jetty-util, servlet-api, sip-api, and also
cipango-dar from the lib/ext folder. In the Rails
registrar project directory, we create
the jars folder. This is where we copy
the required jar files from the Cipango
distribution.

To host the Rails generated Web
interface, we also need the JRuby-Rack
library [25]. This library was developed
specifically to enable the hosting of
JRuby Web applications, inside Java
EE-compliant servers. We use the
JRuby-Rack library to connect the Rails
generated code with the embedded
Cipango server. The library is provided
in a single jar file, which we obtain
from its Web site [25]. We copy the
downloaded JRuby-Rack jar file to the
jars directory of the registrar
project.
[1] require 'java'
[2] Dir.glob("jars/*.jar").each { |jar| require jar }
[3] require 'my_factories'
[4] require 'my_sip_servlet'
[5] ...
... < include the necessary Java classes > ...
[19]
[20] cipango = Server.new
[21] context_collection = SipContextHandlerCollection.new
[22] cipango.handler = context_collection
[23]
... < setup TCP and UDP connectors > ...
[29]
[31] context.resource_base = "."
[32] context.context_path = "/
[33] context.add_servlet( ServletHolder.new( DefaultServlet.new), "/*" )
[34] context.init_params = HashMap.new(
[36]   'rackup' => File.read('config.ru'),
[37]   'jruby.max.runtimes' => 1
[38] )
[40] context.add_event_listener( MyListener.new )
[41] my_sip_servlet = MySipServlet.new
[42] context.add_event_listener( my_sip_servlet )
[43] context.add_sip_servlet( SipServletHolder.new(my_sip_servlet) )
[44] context_collection.add_handler(context)
[45] cipango.start
[46] cipango.join

Figure 7. Ruby application which embeds the Cipango server.

5.3 Creating the Converged Application

To glue the entire project together, we create the JRuby application in a file called 'application.rb', which we save in the top level directory of the Rails registrar project. The application code is shown in Figure 7.

In line 2 of the Figure 7 listing, in order to gain access to the Java classes, we require all the jar files stored in the jars directory relative to the current path. We include the Java classes from their packages in lines 5-19, which are omitted from the listing for simplicity. We later explain the purpose and content of 'my_factories.rb' and 'my_sip_servlet.rb', the files we require in lines 3 and 4.

In lines 20-22, we start embedding the Cipango engine by creating the Server instance and assigning a handler to the server. The handler processes the SIP and HTTP messages received by the server. Next, we define and configure the two server connectors, the UDP connector for the SIP traffic and the TCP connector for the HTTP traffic, with the corresponding ports 5060 and 3000. For simplicity, we omitted the code of the connectors setup from the listing.

We create the converged application context in line 30, and configure it in lines 31-38. More specifically, in lines 31-32, we set the current directory as the resource base for the converged application, and map the application to the root URL of the server. Next, in line 33 we add the default Jetty servlet to the application context. We use it to
handle the requests for static content of our Web interface. Therefore, in line 35, we initialize the Jetty servlet resource base to the public directory of the registrar project, where we store the static content of the Rails application. We incorporate the entire Web interface, generated with Rails, by adding the servlet filter to the converged context in line 39. We use the servlet filter class called RackFilter, which is provided by the JRuby-Rack library. This servlet filter detects if an HTTP request matches a dynamic URL handled by the Rails generated interface. If the request is matched, the filter uses the Rails framework to process the request. Otherwise, it just passes the request to the other servlets in the converged context. Therefore, if the request is for static content, such as images, css and javascript files, the filter passes the request to the default Jetty servlet, which we previously added to the context.

We now need to modify the default behaviour of the JRuby-Rack library. The library usually operates in the mode where a Rails application is deployed to a Java EE server in a single war archive, together with the complete Ruby on Rails and JRuby distributions. Since there is no JRuby runtime in the server’s JVM, JRuby-Rack creates the runtime and initializes it with the Rails environment. The library then starts the servlet filter operations and connects the created runtime with the Java server using the RackFilter object. Since we are running the Cipango server embedded in a JRuby application, we override the default behaviour and force JRuby-Rack to use the application’s JRuby runtime, instead of creating a new one. We do this in line 40, by providing the converged application context with a custom listener object of the MyListener class. MyListener class is defined in the 'my_factories.rb' file shown in Figure 8.

```ruby
lib/my_factories.rb

```%w[org.jruby.rack.DefaultRackApplicationFactory
     org.jruby.rack.SharedRackApplicationFactory
     org.jruby.rack.RackServletContextListener
     org.jruby.Ruby
].each {|c| include_class c }

class MyFactory<DefaultRackApplicationFactory
  field_accessor :rackContext
  def newRuntime
    runtime = Ruby.get_global_runtime
    $servlet_context=rackContext
    require 'rack/handler/servlet'
    return runtime
  end
end

class MyListener<RackServletContextListener
  field_reader :factory
  def newApplicationFactory(context)
    if factory
      return factory
    else
      return (SharedRackApplicationFactory.new{
        MyFactory.new
      })
    end
  end
end

Figure 8. The code of the 'my_factories.rb' file.

We can see that inside the MyListener class we override the newApplicationFactory method, inherited from the RackServletContextListener Java class. We implement this method to return a custom factory object, which is used for JRuby runtime creation when the converged application context is initialized. The custom factory object is of the MyFactory class. We define this class inside the same file. In this class we only override the inherited newRuntime method. We change the method so that, instead of creating and returning a brand new JRuby runtime,
it returns the current runtime in which the application is already running. Furthermore, we perform the necessary initialization of the application’s runtime to prepare it for the RackFilter operations.

In line 36, we configure the JRuby-Rack library to load 'config.ru', a Web application configuration file generated by Rails in the top level directory of the project.

To handle the SIP traffic, we create and add a SIP servlet object to the converged application context, in lines 41-43 of the Figure 7 listing. This object is an instance of the MySipServlet class, which we define in 'my_sip_servlet.rb' file, shown in Figure 9.

Finally, in line 44 of the Figure 7 listing, we add the application context to context_collection of the server, and start the server in line 45.

SIP functionality is implemented in the Ruby SIP servlet class MySipServlet, shown in Figure 9.

In the Ruby SIP servlet, we define two methods: doRegister and doInvite.

By defining the doRegister method of the SIP servlet, we enable the processing of SIP REGISTER requests. Inside the method, from the received SIP request, we first extract the user name, as well as the remote IP address and port of the SIP client the request is sent from. Next, by using the Rails generated model class SipUser, we query the system database for a valid user with the extracted user name. If we find the user in the database, we return the 200 'OK' response, meaning that the current

```ruby
%w[
  javax.servlet.sip.SipServlet
  javax.servlet.sip.SipSessionListener
].each { |c| include_class c }

class MySipServlet < SipServlet
  include SipSessionListener

  def doRegister(request)
    username = request.from.uri.user
    address = request.remote_addr
    port = request.remote_port
    remote_uri = "sip:#{username}@#{address}:#{port}"

    user = SipUser.find_by_user_name(username)
    if user
      exp = request.get_header('Expires')
      if !exp
        c = request.get_header('Contact')
        c.grep(/expires=(\d+)/)
        exp = $1
      end
      exp = exp.to_i
      if exp == 0
        reg = Registration.find_by_sip_user_id(user.id)
        reg.destroy if reg
      else
        reg = Registration.find_or_create_by_sip_user_id_and_location(user.id, remote_uri)
        reg.location = remote_uri
        reg.save
      end
      request.create_response(200).send
    else
      request.create_response(404).send
    end
  end

  def doInvite(req)
    username = req.get_to.get_uri.get_user
    user = SipUser.find_by_user_name(username)
    reg = user.registration
    if reg
      factory = servlet_context.get_attribute('javax.servlet.sip.SipFactory')
      uri = factory.create_uri(reg.location)
      req.proxy.proxy_to(uri)
    else
      req.create_response(404).send
    end
  end

  def sessionCreated(event)
    ... < handle the event > ...
  end

end
```

Figure 9. Ruby SIP servlet which acts as a SIP registrar and proxy server.
user has successfully registered, and save his location to the database using the Registration model. Otherwise, we send the 404 ‘Not found’ (User not found) response. When a user registers to this registrar, the binding is kept in the database until the explicit request for removal is received or the user is deleted through the Web interface.

According to the RFC 3261 [9], a binding removal request is performed by sending a SIP REGISTER message with the expires parameter of the Contact header field value or the Expires header value specified as 0. Our example application checks these values and performs the binding removal when it is appropriate.

For simplicity, we omit other operations a normal registrar server would perform, such as user authentication and registration expiration. Additionally, we ignore the domain part of the user’s SIP URI.

In the doInvite method we implement the proxy behaviour of our application. When the INVITE SIP request is received, we first extract the user name of the recipient and check for its binding. If the binding is found, the request is proxied to the user’s location, as shown in Figure 10. Otherwise, the response 404 (User not found) is sent back as a response to the original request, as shown in Figure 11. For example, if Alice wants to talk to Bob, she would call him from her SIP client using Bob’s user name. SIP proxy would consult the location database to check for his binding and proxy the request to Bob’s current location.

We can execute the application using the following command:

```ruby
$ jruby application.rb
```

Once the application is fully initialized and running, we can create, update and delete the users of the registrar server through the Web interface on port 3000, and the URL shown in Figure 6.

Furthermore, we can configure a SIP VoIP client to register with our application and use it as an outbound proxy by providing: a valid user name, port 5060, and the IP address of our application, or a fully qualified host name.
By utilizing the approach presented in this paper, Ruby objects can also be registered as handlers to specific events generated by the server. To demonstrate this feature, as an example, MySipServlet class implements the SIP Servlet API interface SipSessionListener. The Ruby servlet object is registered with the application to receive events in line 42 of the Figure 7 listing. When session creation event is generated by the server, the application invokes the sessionCreated method on the servlet object. Other events can be processed in a similar manner.

6 PERFORMANCE ANALYSIS OF THE DEMO APPLICATION

Using the methodology we presented in [26], the performance analysis is performed for the proxy part of the example application. This proxy server either terminates the call by sending back a 404 Not found message in the case the binding of the callee is not found in the database, or it sends an INVITE request to the callee SIP client if the binding is found, as depicted in Figures 10 and 11.

The performance was done using SIPp [27], a tool released under the GNU GPL licence. It is available on almost all UNIX platforms. Furthermore, there is a port for the Windows operating system.

SIPp is a command line performance testing tool for the SIP protocol. It can act as a user agent client (UAC) or a user agent server (UAS). SIPp establishes and maintains multiple calls by generating standard SIP requests and responses. There are several scenarios integrated directly in the SIPp executable. Additionally, it is possible to create a custom scenario. The custom scenarios are saved in XML files and are, together with other options, passed to SIPp as command line parameters. It is possible to inject data from one or more external CSV files.

For the performance analysis of our demo application, a custom scenario is created. There are two parts of the scenario: the UAC and the UAS part. The UAC part of the scenario will generate calls against the application server, retrieving the callee URIs from the external CSV file. The application will either respond with the 404 response to the UAC or proxy the request to the UAS. The UAS part of the scenario will respond to the proxied INVITE message, according to the algorithms shown in Figure 12.

Figure 12. Algorithms of the UAC and the UAS parts of the scenario
Table 1. Performance test results

<table>
<thead>
<tr>
<th>Elapsed time</th>
<th>Target rate (cps)</th>
<th>Call rate (cps)</th>
<th>Call length (s)</th>
<th>Failed calls (calls per period)</th>
<th>Successful calls (calls per period)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:30</td>
<td>2</td>
<td>1,97</td>
<td>59</td>
<td>0</td>
<td>59</td>
</tr>
<tr>
<td>01:00</td>
<td>3</td>
<td>2,97</td>
<td>89</td>
<td>0</td>
<td>89</td>
</tr>
<tr>
<td>01:30</td>
<td>4</td>
<td>3,97</td>
<td>119</td>
<td>0</td>
<td>119</td>
</tr>
<tr>
<td>02:00</td>
<td>5</td>
<td>4,97</td>
<td>149</td>
<td>0</td>
<td>149</td>
</tr>
<tr>
<td>02:30</td>
<td>6</td>
<td>5,97</td>
<td>179</td>
<td>3</td>
<td>175</td>
</tr>
<tr>
<td>03:00</td>
<td>7</td>
<td>6,97</td>
<td>209</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>03:30</td>
<td>8</td>
<td>7,97</td>
<td>239</td>
<td>28</td>
<td>239</td>
</tr>
<tr>
<td>04:00</td>
<td>9</td>
<td>8,97</td>
<td>269</td>
<td>46</td>
<td>269</td>
</tr>
<tr>
<td>04:30</td>
<td>10</td>
<td>9,97</td>
<td>299</td>
<td>58</td>
<td>299</td>
</tr>
<tr>
<td>05:00</td>
<td>11</td>
<td>10,97</td>
<td>329</td>
<td>70</td>
<td>329</td>
</tr>
<tr>
<td>05:30</td>
<td>12</td>
<td>11,97</td>
<td>359</td>
<td>84</td>
<td>359</td>
</tr>
<tr>
<td>06:00</td>
<td>13</td>
<td>12,93</td>
<td>389</td>
<td>94</td>
<td>389</td>
</tr>
</tbody>
</table>

Performance tests were conducted against the application installed on a dual core AMD based 2.3 GHz desktop machine with 2 GB of RAM. Its responses were measured using the SIPp with the call rates starting from 2 calls per second (cps), and increasing by 1 cps every 30 seconds, up to 13 cps.

The results of the performance tests are presented in Table 1. The table shows that the application could handle requests without the failed calls at the rates of up to 5 calls per second. After that rate the call length starts to increase which is a direct result of slow application server responses. As a consequence, the application cannot handle all requests sent by the UAC, hence failed calls appear.

The registrar and proxy application we presented does not include the authentication and other standard features. But even in this simple form, it can be utilized in real life where these features are not required, such as the VoIP systems in small or middle size organizations.

7 CONCLUSION AND FUTURE WORK

In this paper, we have presented an infrastructure that supports development of full featured converged Web and SIP applications entirely in Ruby. The infrastructure is based on an embedded Cipango SIP/HTTP Servlet Application Server, a lightweight SIP/HTTP server based on the Jetty Web Server. We have presented a simple demo application as an example on how to use this infrastructure. By adding handcrafted Rails code, and redefining inherited servlet methods of the MySipServlet class, we could extend our application further. For instance, we could easily provide a Web page where a user could examine the currently registered users, and start a voice communication, by clicking on the appropriate links.

The Ruby application shown in Figure 7 is flexible enough so that we could use it to serve a different Rails application, without any modifications of the code. Furthermore, by changing the 'config.ru' file loaded in line 36 of Figure 7, we could incorporate a Web interface developed with another Rack-compatible Web framework, such as the Sinatra.

Performance testing results show that even a standard desktop machine is capable of handling enough call rates to be applicable for real life applications.

To handle the SIP signaling in JRuby, we relied on the Java SIP servlet API, as implemented by the Cipango engine. This is adequate for a simple application. However, for the more complex SIP call flows, it would be beneficial to develop, on top of the described infrastructure, a
custom Ruby DSL for SIP handling. This DSL could provide the Ruby language a flexible SIP service support similar to the one the Rails framework provides for Web services. This infrastructure can be used as a good starting base for development of DSLs and other facilities for VoIP application development utilizing the full potential of SIP Servlet API and Ruby programming language.

REFERENCES

3. Tate, B.: Beyond Java. O'Reilly Media (2005)
15. TorqueBox, http://torquebox.org
17. JBoss, http://jboss.org
22. JRuby, http://jruby.org