Extracted from:

Enterprise Integration with Ruby

A Pragmatic Guide

Chapter 4

Low-Ceremony Distributed Applications

Few technologies have changed the IT landscape the way networks have. Today networks are ubiquitous. Some key players claim that the network is the computer. Think about it for a moment: when was the last time you switched on your computer and did not immediately connect to the Internet? When was the last time you started your office PC and did not log in to a server using ssh or telnet?

Yes, it's true: networks have changed the way we think about computers significantly. They've changed the way we think about applications, too. Nowadays, even small applications often depend on distributed architectures where parts of a program are made available using network technologies.

That's obvious on the Internet, where you can freely use services offered by Amazon.com or Google. By sending simple HTTP requests, you get back information about a book, or you can search for news about your favorite football team.

Big companies in the banking or telecommunications business were the first to adapt to the distributed applications paradigm. Because these companies often are spread across different continents, they were motivated to find ways to implement a feature or a function only once and to reuse it wherever possible. Today it's so easy to implement interprocess communication that nearly every company can benefit from using the technology.

You do not have to use heavyweight industry standards such as SOAP or CORBA to make your processes talk to each other. In this chapter we'll show you how to use plain sockets and pure HTTP to separate concerns and to distribute business logic across process boundaries.

4.1 "I'd Rather Use a Socket"

During a conference in 2003 someone asked Robert C. Martin about the future of SOAP. After listening to the answers of his panelists, Uncle Bob replied, "I'd rather use a socket."[1] He is right: more often than not it's sufficient to use some plain text files and a socket instead of huge databases and complex middleware. So, let's start by looking at this simple approach.

Using plain sockets in an efficient and platform-independent manner isn't trivial; handling multithreading issues and the like can become fairly complicated. If you're really interested in the nitty-gritty details of socket programming, look at *Unix Network Programming* [Ste98] and the appendix of *Programming Ruby* [TFH05]. Fortunately, a few off-the-shelf solutions are available in Ruby's standard distribution.

Ruby comes with a class called GServer that helps in the creation of generic TCP servers. Written by John W. Small, GServer deals with stuff such as connection handling and distributing requests to different threads, leaving you to implement the business logic. The unavoidable echo server example that sends back everything it gets looks like this:

File 58

```ruby
require 'gserver'

class Parrot < GServer
  def initialize(host = 'localhost', port = '3333')
    super(port, host)
  end

  def serve(client)
    text = client.gets
    client.puts(text)
  end
end

lora = Parrot.new
lora.start
lora.join
```

[1] http://www.artima.com/weblogs/viewpost.jsp?thread=4846

On line 3 we derive our class Parrot from GServer, initializing it on line 5. The serve(client) method gets called whenever a client connects to our server. After reading what the client has to say, it sends it back as an echo. As you might have guessed already, start() starts the server, and the join() method ensures that all running threads finish their work before shutting down the server. Here we have a recording from a live performance:

```
mschmidt:/tmp> telnet localhost 3333

  ⇒   Trying ::1...
      Connected to localhost.
      Escape character is '^]'.
  ⇐   Do you wanna have a cookie
  ⇒   Do you wanna have a cookie
      Connection closed by foreign host.

mschmidt:/tmp>
```

The PragBouquet Status Monitor

Because building a server with GServer is so easy, we will try to solve a problem that has been around for a long time now at PragBouquet. Nearly every application running at PragBouquet writes a log file, often used for troubleshooting.

Unfortunately, not all of them (to be exact, no two of them) use the same format, and the files are scattered across several file systems. Even if you're lucky and eventually find the file that should contain the information you need to resolve a trouble ticket initiated by one of your best (and probably most choleric) customers, chances are good that it has been overwritten already and no backup is available.

To overcome this situation we will create a status monitor, a TCP server whose only purpose is to centrally store messages sent by PragBouquet applications. Each message belongs to a certain application and is tagged with a severity level (warn, error, fatal).

You might be wondering "Hey, why rewrite syslogd?" but our little server differs from syslogd in a lot of ways: it's used exclusively by PragBouquet applications, we can decide where and how to store our log messages, special actions can be triggered for certain log levels, and it's platform independent—it will run on Unix boxes as well as on Windows PCs.

Release candidate 1 of status monitor V0.0.1b writes messages only to STDOUT. It looks like this:

File 74

```
Line 1  require 'logger'
     -  require 'gserver'
     -
     -  class StatusMonitor < GServer
     5     def initialize(host = '127.0.0.1', port = '3333')
     -        super(port, host)
     -        @level_map = {
     -           'w' => 'warn', 'e' => 'error', 'f' => 'fatal'
     -        }
    10        @logger = Logger.new(STDOUT)
     -     end
     -
     -     # Expects CSV data in the following format:
     -     # level,application,message.
    15     def serve(client)
     -        level, app, msg = client.readline.chomp.split(',', 3)
     -        if @level_map.has_key?(level)
     -           @logger.send(@level_map[level], "#{app}: #{msg}")
     -           client.puts('0')
    20        else
     -           client.puts('1')
     -        end
     -     end
     -  end
    25  sm = StatusMonitor.new
     -  sm.start
     -  sm.join
```

Let's test it using good old telnet:

```
mschmidt:/tmp> telnet localhost 3333

  ⇒  Trying ::1...
     Connected to localhost.
     Escape character is '^]'.
  ⇐  f,billing,Lost connection to payment gateway.
  ⇒  0
     Connection closed by foreign host.

mschmidt:/tmp>
```

Our request has been successful (0 was returned). The corresponding server output looks like this:

```
F, [2005-09-03T16:32:03.952707 #1124] FATAL -- : billing: \
   Lost connection to payment gateway.
```

Line 18 shows a nice Ruby trick; it dynamically invokes a method on our logger object using send(symbol, [, args...]) (to read more about the Logger class, see Section 6.2, *Logging*, on page 260).

By the way, if you don't want to test your GServer objects using old-fashioned manual prodding with telnet, you can easily use ordinary unit tests, too. Thanks to the magic of duck typing (see the sidebar, on the next page), you can pass any object to the serve() method that responds to the readline() and puts() methods. StringIO works well for this sort of task:

File 68

```
require 'test/unit'
require 'stringio'
require 'status_monitor'

class StatusMonitorTest < Test::Unit::TestCase
   def setup
      @server = StatusMonitor.new
   end

   def test_empty_string
      result = simulate_request("\n")
      assert_equal('1', result)
   end

   def test_invalid_level
      result = simulate_request("x,foo,invalid level\n")
      assert_equal('1', result)
   end

   def test_normal_case
      result = simulate_request("e,foo,normal case\n")
      assert_equal('0', result)
   end

   def simulate_request(request)
      client = StringIO.new(request)
      @server.serve(client)
      client.string[request.size - 2 .. -2]
   end
end
```

This produces the following:

```
Loaded suite sm_test
Started
..E, [2005-10-14T08:00:54.960679 #374] ERROR -- : foo: normal case
.
Finished in 0.012851 seconds.

3 tests, 3 assertions, 0 failures, 0 errors
```

For the unit tests, we've initialized StringIO objects with messages that potential clients could send to the status monitor. The Status Monitor's serve() method doesn't care what class its clients belong to and happily reads requests from a StringIO object and writes results to it.

The main work is done by the simulate_request() method. This calls the

Duck Typing

Ruby is an object-oriented language. Where there are objects, classes and types are usually not far away. However, in contrast to Java or C++ programs, Ruby programs are not cluttered with type declarations. In Ruby, the type of variables, methods, and method parameters do not have to be explicitly declared. Despite this, all Ruby objects have a certain type.

However, when you're programming in a dynamic language, you soon realize that the most important question about a particular object is normally not "What's its class, and what are *all* the methods it responds to?" but instead "Does the object at hand respond to foo()?"

That's where the duck analogy comes from: if it walks like a duck and talks like a duck, then it will be treated as if it is a duck. No matter if it actually is one or not.

This situation is not uncommon even in apparently statically typed languages. For example, Java programmers expect every object to have a toString() method. In Java, this is because all objects are derived from the omnipresent Object class. The implementation is different, but the principle is the same: you want a particular object to respond to a certain message; its class doesn't matter much to you.

Our implementation of the GServer's serve(client) method doesn't care about client's class: it doesn't declare its type, and it doesn't check whether it actually is some kind of TCP socket. It expects client to respond only to methods called readline() and puts(text).

serve() method, passing it a StringIO object. To read the result on line 28, we have to ignore the message that is still in the StringIO object, and we have to ignore the line feed the server sends at the end of the message.

Status Monitor Clients

The biggest problem with servers is that they are totally useless without a client. In our case, we have to create one for every programming language PragBouquet uses. Although this may seem like a lot of redundancy, it's always a good idea to provide programmers with the highest degree of flexibility. If there is no status monitor client library for Java, they will not use it (the status monitor, of course. Not Java). If you think the status monitor is a Good Thing and you want your colleagues to use it, then you have to make it as painless as possible to do so.

Currently, there are not many Ruby applications in the PragBouquet environment, but we want to be prepared. Feeding our status monitor from Ruby programs can be accomplished using class TCPSocket:

File 78

```ruby
require 'socket'

class StatusMonitorClient
  WARN = 'w'
  ERROR = 'e'
  FATAL = 'f'

  def initialize(host, port)
    @sm = TCPSocket.new(host, port)
  end

  def warn(app, msg) log(WARN, app, msg); end
  def error(app, msg) log(ERROR, app, msg); end
  def fatal(app, msg) log(FATAL, app, msg); end
  def terminate() @sm.close; end

  private

  def log(level, app, msg)
    @sm.puts [level, app, msg].join(',')
    @sm.readline
  end
end
```

At the heart of class Status MonitorClient is the log(level,app,msg) method. It sends a string to the status monitor and reads back the result. Because TCPSocket objects behave like any other IO instance, it's easy to implement. You can use this class as follows:

File 78

```ruby
sm = StatusMonitorClient.new('127.0.0.1', 3333)
# ...
sm.fatal('billing', 'Lost connection to payment gateway.')
# ...
sm.terminate
```

One of the most important clients will be the Java client, because it has been the programming language of choice for a long time at PragBouquet. As we did with the Ruby client, we won't go abstracting too much. At the same time, adding a little layer on top of the TCP stack will certainly pay off in the future:

File 71

```java
import java.io.*;
import java.net.*;

public class StatusMonitor {
    public static final String WARN = "w";
    public static final String ERROR = "e";
    public static final String FATAL = "f";

    public StatusMonitor(final String host, final int port)
        throws UnknownHostException, IOException
    {
        _sm = new Socket(host, port);
        _out = new PrintWriter(_sm.getOutputStream(), true);
        _in = new BufferedReader(
            new InputStreamReader(_sm.getInputStream())
        );
    }

    public int warn(final String app, final String msg)
        throws IOException
    {
        return log(WARN, app, msg);
    }

    public int error(final String app, final String msg)
        throws IOException
    {
        return log(ERROR, app, msg);
    }

    public int fatal(final String app, final String msg)
        throws IOException
    {
        return log(FATAL, app, msg);
    }

    public void terminate() throws IOException {
        _out.close();
        _in.close();
        _sm.close();
    }

    private int log(
        final String level,
        final String app,
        final String msg) throws IOException
    {
        final String DEL = ",";
        _out.println(level + DEL + app + DEL + msg);
        final String response = _in.readLine();
        int return_code = 1;
        try {
                return_code = Integer.parseInt(response);
            }
            catch(NumberFormatException ignoreMe) {}
            return return_code;
        }

        private Socket _sm;
        private PrintWriter _out;
        private BufferedReader _in;
    }
```

Again, most of the work takes place in the log(level,app,msg) method. Because this is a book about Ruby, we will not go into all the nitty-gritty details of the StatusMonitor class. However, we will demonstrate how to use it:

File 72

```
public class StatusMonitorTest {
    public static void main(String[] args) {
        try {
            final StatusMonitor sm = new StatusMonitor(
                "127.0.0.1",
                3333
            );
            final int result = sm.debug("billing", "ALAAARM!!");
            System.out.println(result);
            sm.terminate();
        }
        catch(Exception e) {
            System.err.println(
                "An error occurred: " + e.getMessage()
            );
        }
    }
}
```

Oh, and we shouldn't forget the ragged hordes of Perl programmers who inhabited our IT department in the past. All the poor guys who have to maintain their legacy code have the right to use our amazing status monitor, too:

File 73

```
package StatusMonitor;

use strict;
use IO::Socket;

use constant WARN => 'w';
use constant ERROR => 'e';
use constant FATAL => 'f';

sub new {
    my $class = shift;
    my ($host, $port) = @_;
    my $self = {};
    my $sm = IO::Socket::INET->new("$host:$port");
    die "Could not connect to status monitor: $!\n" unless $sm;
    $self->{'sm'} = $sm;
    bless $self, $class;
}

sub warn() {
    my ($self, $app, $msg) = @_;
    $self->_log(WARN, $app, $msg);
}

sub error() {
    my ($self, $app, $msg) = @_;
    $self->_log(ERROR, $app, $msg);
}

sub fatal() {
    my ($self, $app, $msg) = @_;
    $self->_log(FATAL, $app, $msg);
}

sub _log() {
    my ($self, $level, $app, $msg) = @_;
    my $sm = $self->{'sm'};
    $sm->print("$level,$app,$msg\n");
    $sm->getline;
}

sub DESTROY {
    close($_[0]->{'sm'});
}

1;
```

For those who are familiar with Perl (the Pathological Eclectic Rubbish Lister, as its inventor Larry Wall sometimes calls it), the previous lines should be no problem. For all the others it will look like line noise until they have studied Perl for several years. Just believe it: this code achieves the same results as our Ruby and Java code. Here's a little test program to prove it:

File 79

```
use strict;
use status_monitor;

my $sm = StatusMonitor->new('127.0.0.1', 3333);
$sm->error('tracking', 'Lost connection to service.');
```

This produces the following:

```
E, [2005-09-17T17:23:43.216794 #1419] ERROR -- : tracking: \
   Lost connection to service.
```

Admittedly, this is a lot of code that has to be maintained from now on, but it has some valuable benefits. Although we had to implement TCP client code for various languages, we can now add features to the status monitor without touching the client libraries. For example, we can send an e-mail to the operations department whenever an application logs a fatal error. Additionally, to change the transport layer, only the different log(level,app,msg) methods have to be changed.

Adding Better Persistence

Let's illustrate that flexibility by replacing our cheap logger by a full-blown MySQL database. At the same time, we'll add an e-mail feature, too. Our little database (called smon) consists of only one table, log_entries. Its structure should be fairly clear:

File 67

```
create table log_entries(
  id int unsigned not null primary key,
  application varchar(64) not null,
  level enum('warn', 'error', 'fatal'),
  message text,
  created timestamp not null
);
```

We'll access the status monitor database using ActiveRecord:

File 75

```
require 'rubygems'
require 'active_record'
require 'gserver'

ActiveRecord::Base.establish_connection(
  :adapter => 'mysql',
  :host => '127.0.0.1',
  :database => 'smon'
)

class LogEntry < ActiveRecord::Base; end

class StatusMonitor < GServer
   def initialize(host = '127.0.0.1', port = '3333')
      super(port, host)
      @level_map = {
         'w' => 'warn', 'e' => 'error', 'f' => 'fatal'
      }
   end

   # Expects CSV data in the following format:
    # level,application,message.
    def serve(client)
      level, app, msg = client.readline.chomp.split(',', 3)
      if @level_map.has_key?(level)
        entry = LogEntry.new
        entry.application = app
        entry.level = @level_map[level]
        entry.message = msg
        entry.save!
        client.puts('0')
      else
        client.puts('1')
      end
    end
  end
```

First, we removed the Logger class (maybe we will use it again for logging the status monitor's own status, but for now we try to keep the examples short). Next, we added initialization code for ActiveRecord and the LogEntry class (to learn more about ActiveRecord, see Section 2.3, *ActiveRecord Basics*, on page 33). In our serve(client) method we didn't have to change a lot either. All the logging stuff was replaced by initialization code for a new LogEntry object that gets saved for each request.

None of the client libraries had to be touched. By changing only a few lines of Ruby code, we solved one of our biggest problems—important information is now stored centrally. From now on we do not have to search tons of log files to find vital information; a simple SELECT statement will be sufficient. Provided that your database and system administrators earn their salaries by creating regular database backups, this information will be safe forever.

Sending E-mails

At this point we're unstoppable. Let's add some more code to the status monitor that sends an e-mail to the operations department whenever an application logs a fatal error:

File 76

```
require 'rubygems'
require 'active_record'
require 'gserver'
require 'tmail'
require 'net/smtp'


ActiveRecord::Base.establish_connection(
  :adapter => 'mysql',
  :host => '127.0.0.1',
  :database => 'smon'
)

class LogEntry < ActiveRecord::Base; end

class StatusMonitor < GServer
   def initialize(host = '127.0.0.1', port = '3333')
      super(port, host)
      @level_map = {
         'w' => 'warn', 'e' => 'error', 'f' => 'fatal'
      }
   end

   # Expects CSV data in the following format:
   # level,application,message.
   def serve(client)
      level, app, msg = client.readline.chomp.split(',', 3)
      if @level_map.has_key?(level)
         entry = LogEntry.new
         entry.application = app
         entry.level = @level_map[level]
         entry.message = msg
         entry.save!

         inform_helpdesk(app, level, msg) if level == 'f'
         client.puts('0')
      else
         client.puts('1')
      end
   end

   private

   def inform_helpdesk(app, level, msg)
      subject = "A fatal error occurred in #{app}!"
      subject << " Regret your sins!"
      mail = create_mail(
         'helpdesk@pragbouquet.com',
         subject,
         msg
      )
      send_mail('localhost', mail)
   end

   def create_mail(to, subject, body)
     mail = TMail::Mail.new
     mail.date = Time.now
     mail.mime_version = '1.0'
     mail.set_content_type 'text', 'plain'
     mail.from = 'status_monitor@pragbouquet.com'
```

```
 -          mail.to = to
 -          mail.subject = subject
 -          mail.body = body
 -          mail
65        end
 -
 -        def send_mail(host, mail)
 -          msg = mail.encoded
 -          Net::SMTP.start(host, 25) do |smtp|
70            smtp.send_mail(msg, mail.from_address, mail.destinations)
 -          end
 -        end
 -    end
```

Sending e-mail can be divided into two parts: creating the SMTP (Simple Mail Transfer Protocol) formatted e-mail and sending that e-mail over a network. To create the SMTP representation, we used Minero Aoki's excellent *tmail* library[2] (you can see how to obtain and install it in Section 6.4, *Deploying with setup.rb*, on page 287).

In the create_mail(to,subject,body) method, we'll make a TMail::Mail object that contains all the attributes we'd expect an e-mail to have. Finally, on line 68, we call encoded(). It returns something like this:

```
Date: Sat, 17 Sep 2005 14:48:30 +0200
From: status_monitor@pragbouquet.com
To: helpdesk@pragbouquet.com
Subject: A fatal error occurred in billing! Regret your sins!
Mime-Version: 1.0
Content-Type: text/plain

Lost connection to payment gateway.
```

Ruby's standard class Net::SMTP is quite handy for sending the e-mail we just created. Lines 69 to 71 of method send_mail(host,mail) show how to accomplish this.

We're done! The application is now backed by a database and sends an e-mail to the operations department whenever a fatal application error occurs.

4.2 Remote Procedure Calls Using HTTP

Our new status monitor has been a huge success—the system failure rate has dropped down to an all-time low of two per day, each lasting

[2] http://raa.ruby-lang.org/project/tmail

for less than ten minutes. Unfortunately, this is true only for the time between 8 a.m. and 5 p.m., Monday to Friday. Every night three to five fatal application errors still occur, and they don't get solved until the following morning.

Obviously, the status monitor e-mails are read only during office hours. It would increase our quality of service significantly if we could inform our operations department's employees of failures wherever they are and whatever time it is. A good way to do this is to use the Short Messages (SMS) supported by cellular networks: to send the message you simply need a cellular number, and to receive it the employee needs only to keep their cell phone with them.

Waking Up the Operator

To send short messages from a computer to a cell phone, you can do the following:

1. Serially connect a computer and a cellular phone or modem that can be controlled via an AT cellular command interface.
2. Connect to a Short Message Service Center (SMSC) at your network provider. You send messages using a protocol called Short Message Peer-to-peer Protocol (SMPP)[3] to the SMSC. The SMSC is responsible for delivering them to the according mobile devices.
3. Use an existing web service on the Internet.
4. Use an e-mail-to-SMS gateway.

PragBouquet chose the SMSC connection. We already have a server that is capable of sending SMS. This particular piece of software offers a simple interface via HTTP: it supports the single function send(recipient, sender, type, data).

You have to distinguish between Short Messages containing textual or binary data. Set type to text or binary accordingly (text is the default). Binary data has to be transmitted in two-digit hexadecimal ASCII representation, so a binary zero is transmitted as 00 and binary 63 is transmitted as 3f.

The remaining parameters contain the data to be sent and the recipient's and the sender's phone numbers in international format. If the sender parameter is not set, it will be set to the recipient parame-

[3] http://en.wikipedia.org/wiki/SMPP

More Privacy!

Even in the early days of the World Wide Web, security was an important issue. In 1994 Netscape Communications invented a protocol called Secure Sockets Layer (SSL)[*] that enabled web clients and servers to exchange sensitive data without worries.

The first protocol version and its implementation were full of flaws, but since then, SSL has become a de facto standard and has been improved several times. Currently, its most popular implementation is OpenSSL.[†] The OpenSSL project has created a C library that has been ported to countless platforms and is the basis for Ruby's SSL support, too.

In the simplest cases it makes nearly no difference whether you use HTTP or HTTPS. For example, printing the index page of the OpenSSL web site can be achieved as follows:

File 94

```ruby
require 'net/https'

h = Net::HTTP.new('www.openssl.org', 443)
h.use_ssl = true
h.get2('/') { |response| print response.body }
```

You have to set the use_ssl attribute of your Net::HTTP object to true, and you have to use get2(path,initheader=nil,&block) instead of get(path,initheader=nil,dest=nil,&block). Oh, and it's important that you explicitly set the SSL port (443) when opening the connection. Otherwise, the HTTP default port (80) will be used, and your request will probably fail.

Because it's based on the OpenSSL reference implementation, Ruby's support of the SSL protocol is as secure and complete as it can be. You can encrypt and decrypt communication transparently (as we did in the previous example). There are methods for signing and verifying both client and server certificates. For example, peer_cert() returns the server's X.509 certificate, and using cert=(certificate) you can set a client's X.509 certificate. The library allows you to maintain a certificate store in your local file system, too.

[*] http://en.wikipedia.org/wiki/Secure_Sockets_Layer
[†] http://www.openssl.org

ter automatically. This way the recipient thinks he has sent himself a message—in our case that is all right.

The service listens on port 4242 under path /send. It returns its result using the HTTP status code, so 200 means everything went fine and 500 means an error occurred. Let's try it using telnet:

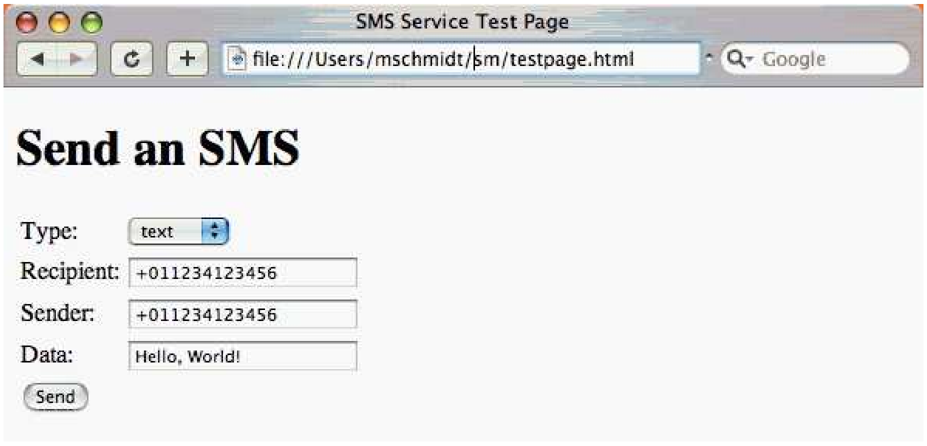
```
mschmidt:/tmp> telnet localhost 4242
  ⇒   Trying ::1...
      Connected to localhost.
      Escape character is '^]'.
  ⇐   GET /send?type=text&recipient=+011234123456&data=hello HTTP/1.0
  ⇒
      HTTP/1.1 200 OK
      Connection: close
      Date: Wed, 14 Sep 2005 20:26:47 GMT
      Content-Type: text/plain
      Server: WEBrick/1.3.1 (Ruby/1.8.2/2004-12-25)
      Content-Length: 0

      Connection closed by foreign host.
mschmidt:/tmp>
```

Hmm, there's no error message from the session, but our cell phone didn't yell "YOU HAVE A NEW MESSAGE!" at full volume as it usually does when it receives an SMS. So, what's wrong? Provided that the SMS server is working properly, the problem must be hidden in our input. Obviously, the type and data parameters are correct, but the recipient's phone number contains a subtle syntax error. All international phone numbers start with one of the following prefixes:[4]

- 00<international area code><national area code>
- +<international area code><national area code>

At first sight, everything seems to be all right with the recipient's phone number, but we forgot that we are using an HTTP service that expects all its parameters in URL-encoded format. Hence, the leading + sign is interpreted as a blank that renders the phone number completely wrong. Instead of transmitting +011234123456 (an American phone number), ␣011234123456 (a national phone number with a leading blank) is transmitted.

[4]Usually, a leading zero of the national area code is omitted, but you should not blindly follow this rule. For Italy, as an example, you have to explicitly transmit the leading zero of the national area code in international phone numbers.



Send an SMS

Type: text
Recipient: +011234123456
Sender: +011234123456
Data: Hello, World!
Send

Figure 4.1: Testing the SMS Server

In the best case, this phone number does not exist, and the SMSC fails to deliver your message. In the worst case, someone could become really angry while you're desperately running your unit test suite over and over again at 2 a.m. on a Saturday night.

When testing complex HTTP services, the telnet command can be a bit tedious. For services expecting GET requests, you're much better off using an ordinary web browser (as shown in Figure 4.1). The browser transparently handles all encoding issues for you (command-line diehards will use wget[5] or curl[6] anyway). An HTML page such as the following is sufficient for testing purposes:

File 80

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html>
   <head>
      <title>SMS Service Test Page</title>
   </head>
   <body>
      <h1>Send an SMS</h1>
      <form action="http://localhost:4242/send">
        <table>
           <tr>
```

[5]http://www.gnu.org/software/wget/wget.html
[6]http://curl.haxx.se

```
                <td>Type:</td>
                <td>
                   <select name="type">
                      <option>text</option>
                      <option>binary</option>
                   </select>
                </td>
             </tr>
             <tr>
                <td>Recipient:</td>
                <td><input type="text" name="recipient"/></td>
             </tr>
             <tr>
                <td>Sender:</td>
                <td><input type="text" name="sender"/></td>
             </tr>
             <tr>
                <td>Data:</td>
                <td><input type="text" name="data"/></td>
             </tr>
             <tr>
                <td colspan="2">
                   <input type="submit" value="Send"/>
                </td>
             </tr>
          </table>
        </form>
     </body>
  </html>
```

Another way to circumvent problems like these is to use a mature HTTP client library, such as the one that comes with the Ruby standard distribution. Let's see how we can perform HTTP requests and encapsulate the SMS server:

File 69

```
require 'net/http'
require 'cgi'

class SmsService
   def initialize(host = '127.0.0.1', port = 4242)
     @host, @port = host, port
   end

   def send_text(params)
      send_sms(params, 'text')
   end

   def send_binary(params)
      send_sms(params, 'binary')
   end

```

```
 -        private
 -
 -        def send_sms(params, type)
20           result = false
 -           Net::HTTP.start(@host, @port) do |http|
 -              query = "type=#{type}"
 -              params.each do |k,v|
 -                 query << "&#{k}=#{CGI.escape(v)}"
25              end
 -              response = http.get("/send?#{query}")
 -              result = response.class == Net::HTTPOK
 -           end
 -           result
30        end
 -     end
```

A small sample program demonstrates the usage of the SmsService class. Finally you (and all your enervated colleagues) can hear it again: “YOU HAVE A NEW MESSAGE!”

File 69

```
sms = SmsService.new
sms.send_text(
  :recipient => '+01123412345678',
  :data => 'Hello, world!'
)
```

In send_sms(params,type) of the SmsService class we open a new HTTP connection using the start(host,port) method of class Net::HTTP. It expects a code block that gets passed the current connection. Within the code block the query string for the GET request is prepared (note that we have to use the encode(string) method of the CGI class to URL-encode our query parameters).

Eventually, on line 26 we initiate a GET request that returns an object derived from Net::HTTPResponse. This object encapsulates everything that makes up an HTTP response. Its most important methods are as follows:

- code() returns the HTTP status code.
- message() returns the HTTP status message.
- body() returns the response body.
- The headers hash contains all HTTP headers.

There is a separate class for every HTTP status code (yes, this little library defines more than 50 classes), so instead of checking whether code() returns 200, you can alternatively check whether your response object’s class is Net::HTTPOK (as we did in line 27).

URL Encoding

In the beginning of computer networking everything had to be represented in ASCII characters using only 7 bits. Because of this, nearly all protocols in today's Internet turn non-ASCII data into 7-bit ASCII characters somehow. RFC 1738* painstakingly explains which characters are allowed in URLs:

"Only alphanumerics [0--9a--zA--Z], the special characters [$-_.+!*'(),], and reserved characters used for their reserved purposes may be used unencoded within a URL."

The rules are simple: all ASCII control characters (0x00–0x1f, 0x7f), all non-ASCII characters (0x80–0xff), and all reserved characters ([$&+,/:;=?@]) have to be encoded under all circumstances.

In addition, some characters considered unsafe should be encoded, too: ["?<>#%{}|\^~`] and the square brackets ([,]) themselves (0x5b, 0x5d).

Encoding a single value is easy: you prepend its case-insensitive, two-digit hexadecimal ISO-Latin code by a percent symbol (%).

For example, the blank character is encoded as %20 (for convenience, a blank can also be encoded as a single '+' symbol) and an uppercase 'A' (whose ISO-Latin code is 65) is turned into %41.

*http://www.rfc-editor.org/rfc/rfc1738.txt

Integrating the SMS service into StatusMonitor is left as an exercise for the reader. Keep in mind that short messages do have to be short: they shouldn't be more than 160 characters.

Sending Java stack traces is definitely not an option.

If you don't like encoding query parameters manually, you can use HTTP's POST command. In the current example it doesn't make a big difference, though. Here's how send_sms(params,type) would look:

File 70

```
def send_sms(params, type)
  result = false
  Net::HTTP.start(@host, @port) do |http|
    query = "type=#{type}"
    params.each { |k,v| query << "&#{k}=#{v}" }
```