FULL STACK WEB DEVELOPER PART XIII

Client Server Programming

communication of remote processes

Panos M.

www.techcareerbooster.com

Full Stack Web Developer Part XIII: Client Server Programming

Panos Matsinopoulos

This book is for sale at
http://leanpub.com/full-stack-web-developer-part-xiii-client-server-programming

This version was published on 2019-08-19

Contents

The Bundle and the TCB Subscription

Part of Bundle

This book is not sold alone. It is part of the bundle Full Stack Web Developer.

Goes with a TCB Subscription

When you purchase the bundle, then you have full access to the contents of the TCB Courses.

Each TCB Subscription Goes with the Bundle

Moreover, this goes vice-versa. If you purchase the subscription to the TCB Courses, then you are automatically eligible for the Full Stack Web Developer bundle.

Credits To Photo on Cover Page

Image by fernando zhiminaicela from Pixabay

About and Copyright Notice

Full Stack Web Developer - Part XIII - Client Server Programming 1st Edition, August 2019

by Panos M. for Tech Career Booster (https://www.techcareerbooster.com)

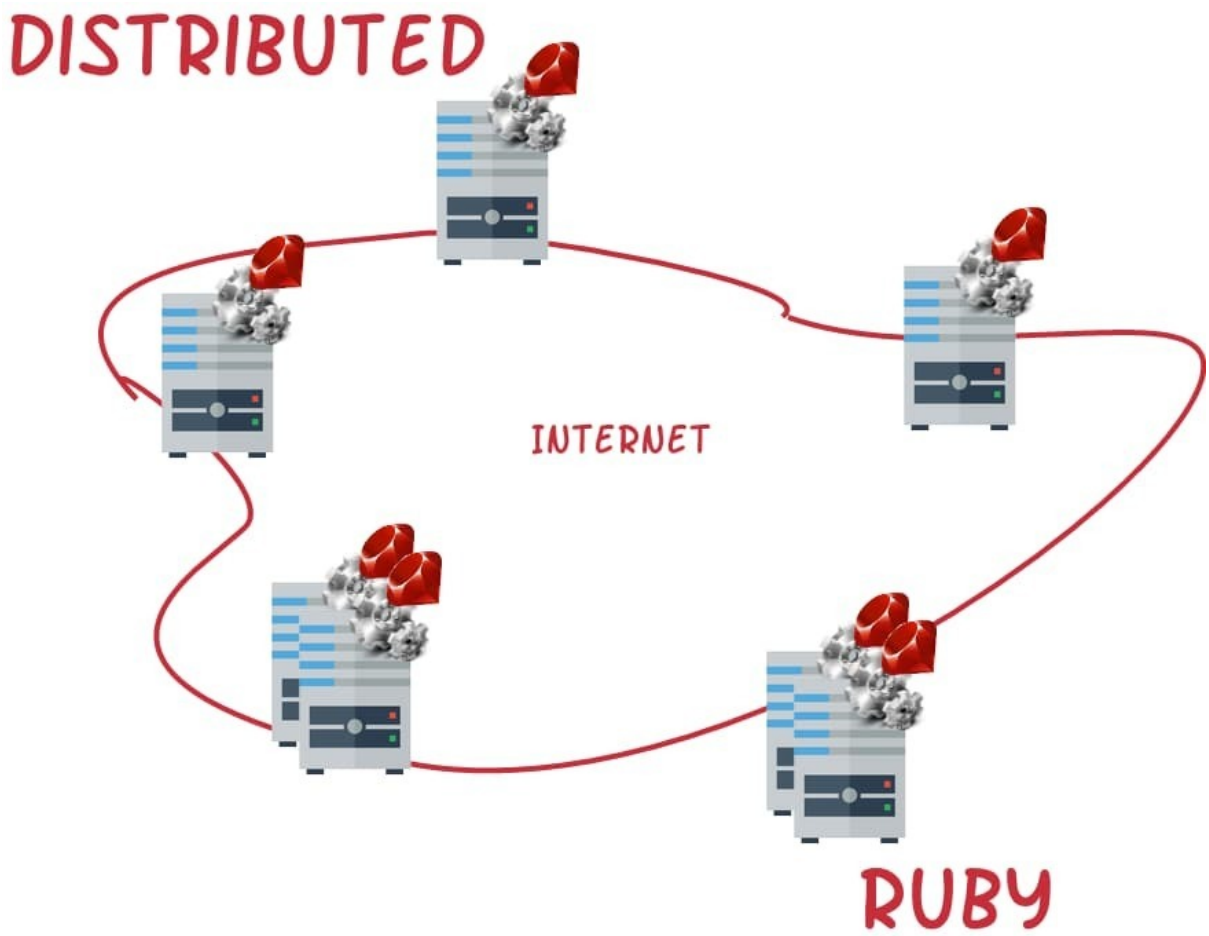
For more information: https://www.techcareerbooster.com

Client Server Programming

In this book, we are going to learn how processes, distributed and running in different machines, exchange messages and how Internet protocols make all this very easy. The main subjects of this section are Distributed Ruby and TCP/IP sockets programming. Other books that follow, rely on the knowledge that you acquire here, and will finally turn you into a Junior Web Developer!

1 - Distributed Ruby

Summary



Distributed Ruby

This is the chapter that introduces you to Web development. I.e. development of software that is deployed in Internet and uses its technologies. We are learning how to design and deploy applications that follow the client-server architecture paradigm. We rely on the protocol dRuby, Distributed Ruby, which is coming for free with the Ruby standard library. We will also have an introduction to internet protocols.

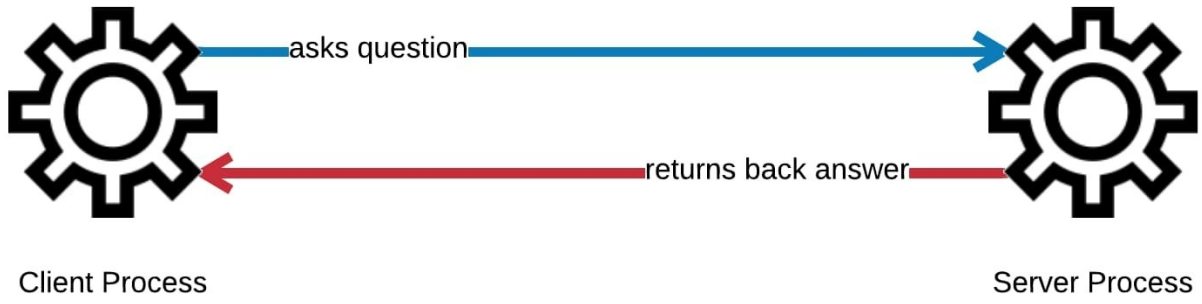
Learning Goals

1. Learn about the client-server architecture paradigm.
2. Learn about the protocols of the TCP/IP family.
3. Learn about distributed Ruby.
    1. How you can write the server code.
    2. How you can write the client code.
    3. How the method is invoked on the local and on the remote objects.

    4. How client code treats classes that it knows and classes that it does not know about.
    5. How and under which conditions a method called on a DRb object is forwarded to the remote object.
4. Learn about IP addresses.
5. Learn about domain name servers.
6. Learn about the ports.
7. Learn about the well-known ports.
8. Learn how you can remove a method definition from an object.
9. Learn about the tainted variables.
10. Learn about the different Ruby safe levels.

Introduction

We have done lots of Ruby programming in the previous chapters but we didn't exploit the power of the network. We stayed within the confines of a single Ruby process, although we have talked about multi-process programming. However, there is another architecture paradigm in software engineering which is called *client-server* architecture. In this paradigm, a process plays the role of the *client* asking questions and another one plays the role of the *server* returning back the answers.



Client Server Programming

Although you have not programmed such an application, that is split in many processes, clients and servers, you have definitely used applications that rely on this model. For example, the browser works like that. It is the client side of the communication. It sends requests to the Web server, which is the *server* side of this architecture. Web server returns back the response and the browser is responsible to render it so that the user can see.

Another example is `mysql` command line application that communicates with MySql server.

About Protocols

In the software engineering world, we have had various development around this area. Different technologies have been invented to allow a process, possibly written in a programming language X, communicate with another process, possibly written in a different programming language Y and living/running in another computer.

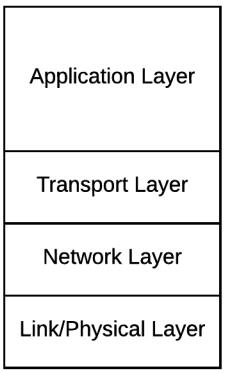
- RPC (Remote Procedure Calls)

- CORBA
- DCOM
- RMI

and many more.

In this course we will focus on the technologies around the family of TCP/IP protocols. TCP/IP (Transmission Control Protocol/Internet Protocol) is the set of rules and protocols that allows two distributed processes to communicate using *Internet*.

TCP/IP family of protocols has been divided into four layers. Special technical groups have specified the rules and details of all these four layers:



TCP/IP Layers

The two most important layers for us, on our way to become Junior Web Developers, are the first two. The Application and the Transport layer.

In the Application Layer one can find popular protocols such as:

1. HTTP/S, which is the protocol that browsers use to exchange messages with the Web servers.
2. SMTP, which is the protocol that email uses.
3. S/FTP, which is a File Transfer Protocol
4. POP3 and IMAP are other email related protocols.

We will focus on the HTTP/S protocol of course. Reference to some other protocols will be done too.

Below the Application layer protocols, TCP (Transmission Control Protocol) is the most important for us now. It is the protocol that allows an application (such as HTTP) to exchange reliable messages with another process over Internet. Another common protocol of this layer is UDP (User Datagram Protocol). UDP is unreliable connectionless protocol, but it is also very efficient in terms of performance.

Before we start doing some work with TCP/IP protocols, we will introduce you to a tool called DRb (Distributed Ruby) which comes in the Ruby Standard Library.

Distributed Ruby - DRb - dRuby

dRuby is a distributed object system for Ruby. This basically means that a process X can call a method on a remote object that lives in another process Y, even if that process Y is on another machine. The process X is the client and the process Y is the server in this client-server model.

Let's start with an example. Create the RubyMine Ruby project `drb-example1`.

Server Code

We will first create the server process code. Create the file `server.rb` inside the root folder of your project:

```
1  # File: drb-example1/server.rb
2  #
3  require 'drb/drb'
4
5  URI = "druby://localhost:8787"
6
7  class TimeServer
8    def get_current_time
9      Time.now
10   end
11 end
12
13 SERVER_OBJECT = TimeServer.new
14
15 puts 'Starting server ....that can tell the time...(Hit Ctrl+C to terminate me)'
16 puts "(I am here: #{URI})..."
17
18 DRb.start_service(URI, SERVER_OBJECT)
19
20 # Wait for the drb server thread to finish before exiting.
21 DRb.thread.join
```

(the above code snippet online)

The things that you need to be aware of are:

1. We need to require `drb/drb` (line 3) because this is part of the Ruby standard library (and not the core library)
2. Each server needs to listen on particular host and port. On our example, we declare this to be the `localhost` host, which refers to the local machine the server is started on, and the port being number `8787`. It could have been any available port in the system we spawn our server on. (See the note about ports later on.)

3. We said that dRuby is a distributed object system. That's why we declare a class and we create an instance of the object that will play the role of the server. This happens in lines 5 up to 13.
4. Then on line 18, we start the `DRb` server, i.e. we tell `DRb` which object is going to serve requests and behind which `URI`, i.e. behind which host and port. This happens on line 18.
5. The server serving requests will be done, internally, by another thread. The parent thread, needs to join that thread before exiting. This is the line 21.

Before we talk about the client code that will call the services of this object, let's try to run the server.

```
1  drb-example1 > $ ruby server.rb
2  Starting server ....that can tell the time...(Hit Ctrl+C to terminate me)
3  (I am here: druby://localhost:8787)...
```

(the above code snippet online)

As you can see the process starts and stays alive. If you want to terminate it, you will need to hit the key combination <kbd>Ctrl + C</kbd>.

Client Code

Let's now write the client code. This is what you have to create:

```ruby
1  # File: drb-example1/client.rb
2  #
3  require 'drb/drb'
4
5  SERVER_URI = "druby://localhost:8787"
6
7  puts 'I am a client and I want to know the current time...'
8  puts "...I am going to ask #{SERVER_URI}"
9
10 timeserver = DRbObject.new_with_uri(SERVER_URI)
11
12 puts "...got back this: #{timeserver.get_current_time}"
```

(the above code snippet online)

1. First we require `drb/drb`, like we did for the server code. (line 3)
2. We declare the endpoint of the server. Otherwise, we will not be able to connect to that server. (line 5)
3. Then, on line 10, we instantiate a local representation of the remote object using the initializer `DRbObject.new_with_uri(...)`. The `timeserver` local variable now has everything we need to call methods on the remote object.

4. This is what we do on line 12. We call the method `#get_current_time` which is not implemented in the client code of course. It is implemented in the remote server code.

All the technical details to transfer the request from the client process to the server process and get back the result is being handled magically by `drb`.

Let's run the client code to see it in action. Open another terminal, while on the original terminal you have the server code still running. Execute the command that is given below:

```
1 drb-example1 > $ ruby client.rb
2 I am a client and I want to know the current time...
3 ...I am going to ask druby://localhost:8787
4 ...got back this: 2017-04-21 13:54:39 +0300
5 drb-example1 > $
```

(the above code snippet online)

You can see how the current date and time are being printed. This is the response as was returned by the server code.

Pretty cool! Isn't it?

- IPs: IP addresses uniquely identify a machine connected to the Internet. This is an example of an IP address: `23.23.128.123`
- DNS: Domain Name Service translates names to IP addresses. Hence, we don't have to remember the IP address of a host machine. We can always remember its name. The translation from name to IP will be done automatically by underlying protocols and via the DNS servers. Try, for example, the command `host www.techcareerbooster.com`. You will get the IP address of this name. Or you can try visiting What is my ip Web site, which will print your IP address.
- Ports: Since we may have, and we do want to have, many different servers running on the same machine, we are using *ports* to allow for multiple servers to be able to listen for multiple clients at the same time. Hence when a server is spawned inside a host machine, it needs to declare which port to occupy. The system will allocate the requested port to this server, until the server program terminates. If the port is not available when server requests for it, system will raise an error and server will not be installed. Note that there are some well-known ports that are used by well-known servers. For example:
  - Port 80: It is used for HTTP traffic. In fact it is the default port Web browsers use when we ask of an address.
  - Port 443: It is used for HTTPS traffic. The Secure, encrypted version of HTTP. It is the default port when we ask our browser to bring a page using `https://` instead of `http://`.
  - Port 7: It is the ECHO server, which returns back whatever client sends to it.
  - Port 21: FTP, File Transfer Protocol
  - Port 22: SSH, Secure Shell protocol
  - Port 23: Telnet protocol
  - Port 25: Simple Mail Transfer Protocol (SMTP)

- Port 110: POP3 protocol, which is used to retrieve email from email boxes.
- Port 143: IMAP protocol, which is another protocol to manage email boxes on an email server.

Remote Objects As Method Results

Calling a method on a remote object that we have constructed using `DRbObject.new_with_uri` was quite easy. The local object representation is done using an instance of the class `DRbObject`.

You can confirm that with the following `drb-example1/client.rb` code:

```
1  # File: drb-example1/client.rb
2  #
3  require 'drb/drb'
4
5  SERVER_URI = "druby://localhost:8787"
6
7  puts 'I am a client and I want to know the current time...'
8  puts "...I am going to ask #{SERVER_URI}"
9
10 timeserver = DRbObject.new_with_uri(SERVER_URI)
11
12 puts "...got back this: #{timeserver.get_current_time}"
13
14 puts "...timeserver is #{timeserver.inspect}"
```

(the above code snippet online)

This is exactly like the first version, but we have added the line 14 to print the inspection details of the `timeserver` variable.

Here is what we see when we run this version of the client code (make sure that `drb-example1/server.rb` is running on another terminal):

```
1  drb-example1 > $ ruby client.rb
2  I am a client and I want to know the current time...
3  ...I am going to ask druby://localhost:8787
4  ...got back this: 2017-04-24 21:19:36 +0300
5  ...timeserver is #<DRb::DRbObject:0x007ffbdb10b420 @uri="druby://localhost:8787", \
6   @ref=nil>
7  drb-example1 > $
```

(the above code snippet online)

You can tell from the last line of the output, that the object returned from `DRbObject.new_with_uri` is a `DRb::DRbObject` instance.

This class allows the client code to call methods on the remote object.

Local Representation of Remote Object

Let's enhance the `drb-example1/client.rb` program to print the class of the result of the method call `timeserver.get_current_time` that takes place on line 12.

Here is the new version:

```
1  # File: drb-example1/client.rb
2  #
3  require 'drb/drb'
4
5  SERVER_URI = "druby://localhost:8787"
6
7  puts 'I am a client and I want to know the current time...'
8  puts "...I am going to ask #{SERVER_URI}"
9
10 timeserver = DRbObject.new_with_uri(SERVER_URI)
11
12 timeserver_result = timeserver.get_current_time
13
14 puts "...got back this: #{timeserver_result}"
15
16 puts "...timeserver is #{timeserver.inspect}"
17 puts "...timeserver_result class is #{timeserver_result.class}"
```

(the above code snippet online)

On line 12, we save the remote method call result into its own variable. Then, on line 17, we print the class of this variable. Here is what we get if we run `drb-example1/client.rb` again:

```
1  drb-example1 > $ ruby client.rb
2  I am a client and I want to know the current time...
3  ...I am going to ask druby://localhost:8787
4  ...got back this: 2017-04-24 21:23:56 +0300
5  ...timeserver is #<DRb::DRbObject:0x007faffb837518 @uri="druby://localhost:8787", \
6   @ref=nil>
7  ...timeserver_result class is Time
8  drb-example1 > $
```

(the above code snippet online)

Do you see that the class returned is `Time`? Matches the class of the result returned by the `TimeServer#get_current_time` (see `server.rb:9`).

Keep this at the back of your head: Server builds and returns a `Time` instance, and client code has this instance locally as a `Time` instance too.

Class Returned Is not Known to Client Namespace

What will happen if server didn't actually return back an instance of a class that is known to the client namespace? Let's change the server code as follows:

```
1  # File: drb-example1/server.rb
2  #
3  require 'drb/drb'
4
5  URI = "druby://localhost:8787"
6
7  class MyTime
8    def initialize(a_time_instance)
9      @a_time_instance = a_time_instance
10   end
11
12   def to_s
13     @a_time_instance.strftime("%Y%m%d%H%M%S")
14   end
15 end
16
17 class TimeServer
18   def get_current_time
19     MyTime.new(Time.now)
20   end
21 end
22
23 SERVER_OBJECT = TimeServer.new
24
25 puts 'Starting server ....that can tell the time...(Hit Ctrl+C to terminate me)'
26 puts "(I am here: #{URI})..."
27
28 DRb.start_service(URI, SERVER_OBJECT)
29
30 # Wait for the drb server thread to finish before exiting.
31 DRb.thread.join
```

(the above code snippet online)

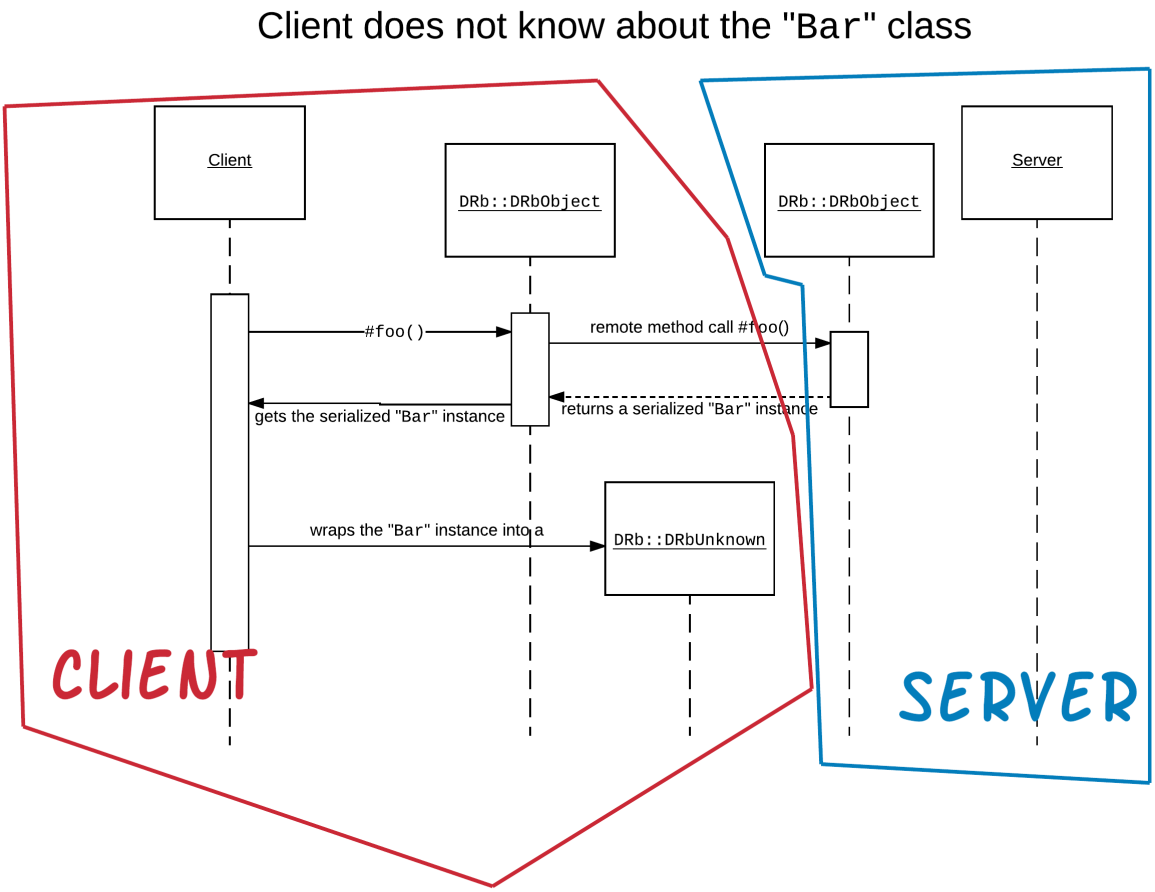
Here is the difference to the previous program. The `TimeServer#get_current_time` now builds an instance of the class `MyTime`. And this is what is returned back to the client.

However, the client code does not know about this class (whereas the client code knew about the `Time` class as it is part of core Ruby). What will the client print if we now run the program? (make sure you restart your server so that it takes the new code into account).

```
drb-example1 > $ ruby client.rb
I am a client and I want to know the current time...
...I am going to ask druby://localhost:8787
...got back this: #<DRb::DRbUnknown:0x007f96229b6670>
...timeserver is #<DRb::DRbObject:0x007f96229b7480 @uri="druby://localhost:8787", \
 @ref=nil>
...timeserver_result class is DRb::DRbUnknown
drb-example1 > $
```

(the above code snippet online)

The result returned back from `#get_current_time` is wrapped into an instance of an object of class `DRb::DRbUnknown`. This wrapping is done automatically by DRb when the class is not known to the client namespace.



in that case Client does not have too many options with the object returned (`DRb::DRbUnknown`)

When Client Does Not Know About The Class Of the Object Returned

Let's print a more informative message about the instance that we have at our hands. Here is the new version of the client code:

```
1  # File: drb-example1/client.rb
2  #
3  require 'drb/drb'
4
5  SERVER_URI = "druby://localhost:8787"
6
7  puts 'I am a client and I want to know the current time...'
8  puts "...I am going to ask #{SERVER_URI}"
9
10 timeserver = DRbObject.new_with_uri(SERVER_URI)
11
12 timeserver_result = timeserver.get_current_time
13
14 puts "...got back this: #{timeserver_result}"
15
16 puts "...timeserver is #{timeserver.inspect}"
17 puts "...timeserver_result class is #{timeserver_result.class}"
18 puts "...timeserver_result inspect is #{timeserver_result.inspect}"
```

(the above code snippet online)

The only new line here is the last one, line 18. We print the `#inspect` result called on the `DRb::DRbUnknown` object. Here is what we get if we run the new version of the `drb-example1/client.rb` code:

```
1  drb-example1 > $ ruby client.rb
2  I am a client and I want to know the current time...
3  ...I am going to ask druby://localhost:8787
4  ...got back this: #<DRb::DRbUnknown:0x007fbc6195a6d0>
5  ...timeserver is #<DRb::DRbObject:0x007fbc6195b4e0 @uri="druby://localhost:8787",\
6   @ref=nil>
7  ...timeserver_result class is DRb::DRbUnknown
8  ...timeserver_result inspect is #<DRb::DRbUnknown:0x007fbc6195a6d0 @name="MyTime"\
9  , @buf="\x04\bo:\vMyTime\x06:\x15@a_time_instanceIu:\tTime\r\x120\x1D\x80\xF2\x11\
10 \xCC\xA0\a:\voffseti\x020*:\tzoneI\"\tEEST\x06:\x06EF">
11 drb-example1 > $
```

(the above code snippet online)

Interesting! the `DRb::DRbUnknown` is wrapping an instance of type `MyTime`. We can tell that from the `@name` value. The `DRb::DRbUnknown#name` method call will return the constant that corresponds to the instance that is wrapped. Also, the `@buf` contains the actual binary representation of the instance.

Unfortunately, in this case, client code needs to give some help to DRb to tell it how to reconstruct the actual `MyTime` instance.

Client Knows The Class Returned Back From Server

Let's create the file `drb-example1/my_time.rb` and put the definition of this class there:

```ruby
# File: drb-example1/my_time.rb
#
class MyTime
  def initialize(a_time_instance)
    @a_time_instance = a_time_instance
  end

  def to_s
    @a_time_instance.strftime("%Y%m%d%H%M%S")
  end
end
```

(the above code snippet online)

Then require this file both at the `server.rb` code and at the `client.rb` code.

Here is the new `server.rb` version:

```ruby
# File: drb-example1/server.rb
#
require 'drb/drb'
require_relative 'my_time'

URI = "druby://localhost:8787"

class TimeServer
  def get_current_time
    MyTime.new(Time.now)
  end
end

SERVER_OBJECT = TimeServer.new

puts 'Starting server ....that can tell the time...(Hit Ctrl+C to terminate me)'
puts "(I am here: #{URI})..."

DRb.start_service(URI, SERVER_OBJECT)

# Wait for the drb server thread to finish before exiting.
DRb.thread.join
```

(the above code snippet online)

And here is the new version of the `client.rb`:

```
1  # File: drb-example1/client.rb
2  #
3  require 'drb/drb'
4  require_relative 'my_time'
5
6  SERVER_URI = "druby://localhost:8787"
7
8  puts 'I am a client and I want to know the current time...'
9  puts "...I am going to ask #{SERVER_URI}"
10
11 timeserver = DRbObject.new_with_uri(SERVER_URI)
12
13 timeserver_result = timeserver.get_current_time
14
15 puts "...got back this: #{timeserver_result}"
16
17 puts "...timeserver is #{timeserver.inspect}"
18 puts "...timeserver_result class is #{timeserver_result.class}"
19 puts "...timeserver_result inspect is #{timeserver_result.inspect}"
```

(the above code snippet online)

Restart your server and then run your client again:

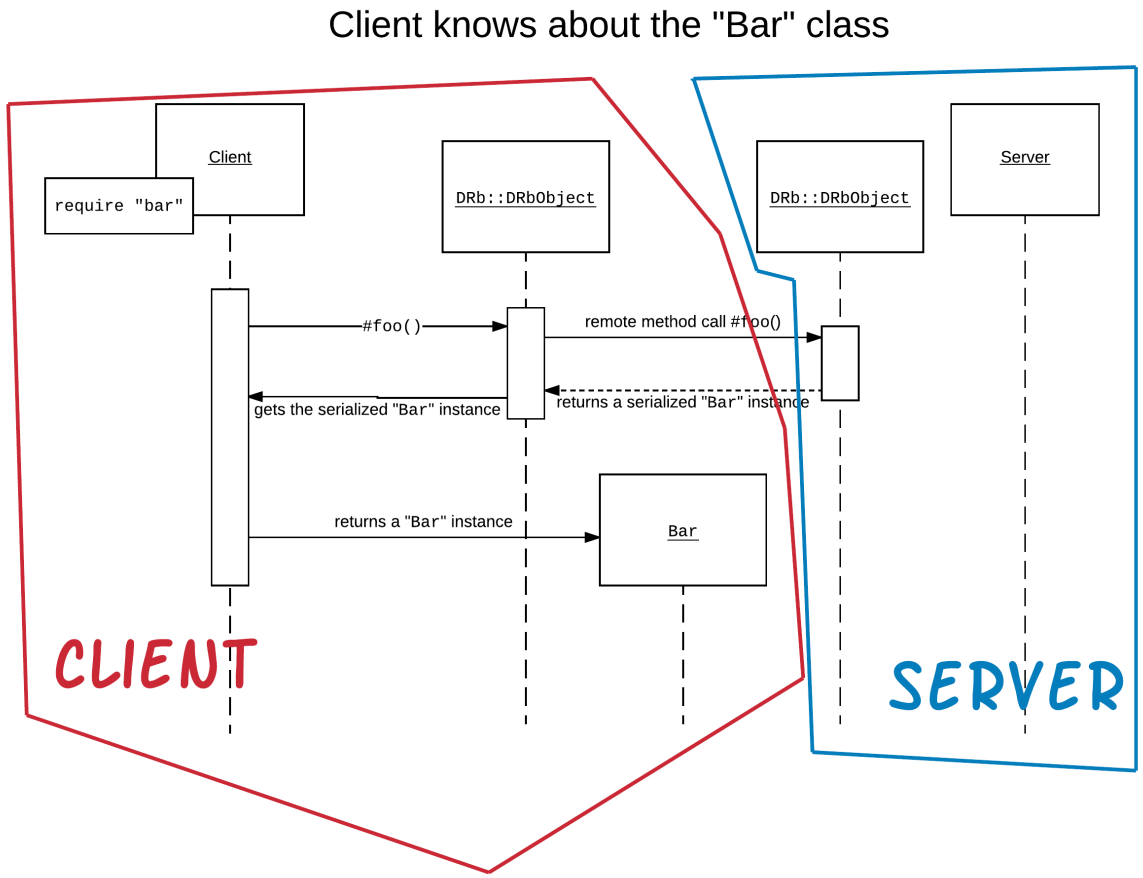
```
1  drb-example1 > $ ruby client.rb
2  I am a client and I want to know the current time...
3  ...I am going to ask druby://localhost:8787
4  ...got back this: 20170424221517
5  ...timeserver is #<DRb::DRbObject:0x007fb7621b1c70 @uri="druby://localhost:8787",\
6   @ref=nil>
7  ...timeserver_result class is MyTime
8  ...timeserver_result inspect is #<MyTime:0x007fb7621b0f28 @a_time_instance=2017-0\
9  4-24 22:15:17 +0300>
10 drb-example1 > $
```

(the above code snippet online)

Bingo! The result returned from `#get_current_time` is no longer wrapped in `DRb::DRbUnknown` instance. It is now a proper `MyTime` instance. And the `...got back this: 20170424221517` verifies that the correct method `#to_s` has been called (line 15).

The question that one might now ask is the following. The `#to_s` method call has been executed in the server process (like the `timeserver.get_current_time` did) or in the local client process?

It has been executed in the local client process. It has been called in the local `MyTime` instance that has been reconstructed by the DRb system.



When Class is Known Calls Are Made to Local Copies

We can confirm that by sleeping for 10 seconds before actually calling the `puts "...got back this: #{timeserver_result}"` command. While sleeping, we can kill server and we will see that the client code will be executed without problem, confirming that the method call is done in the local, and not in the remote, process.

Here is the version of client code with the 10 seconds sleep before the statement that calls the `MyTime#to_s` method:

```
1  # File: drb-example1/client.rb
2  #
3  require 'drb/drb'
4  require_relative 'my_time'
5
6  SERVER_URI = "druby://localhost:8787"
7
8  puts 'I am a client and I want to know the current time...'
9  puts "...I am going to ask #{SERVER_URI}"
10
```

```
11 timeserver = DRbObject.new_with_uri(SERVER_URI)
12
13 timeserver_result = timeserver.get_current_time
14
15 puts "Sleeping for 10 seconds to allow server to terminate..."
16
17 sleep 10
18
19 puts "...got back this: #{timeserver_result}"
20
21 puts "...timeserver is #{timeserver.inspect}"
22 puts "...timeserver_result class is #{timeserver_result.class}"
23 puts "...timeserver_result inspect is #{timeserver_result.inspect}"
```

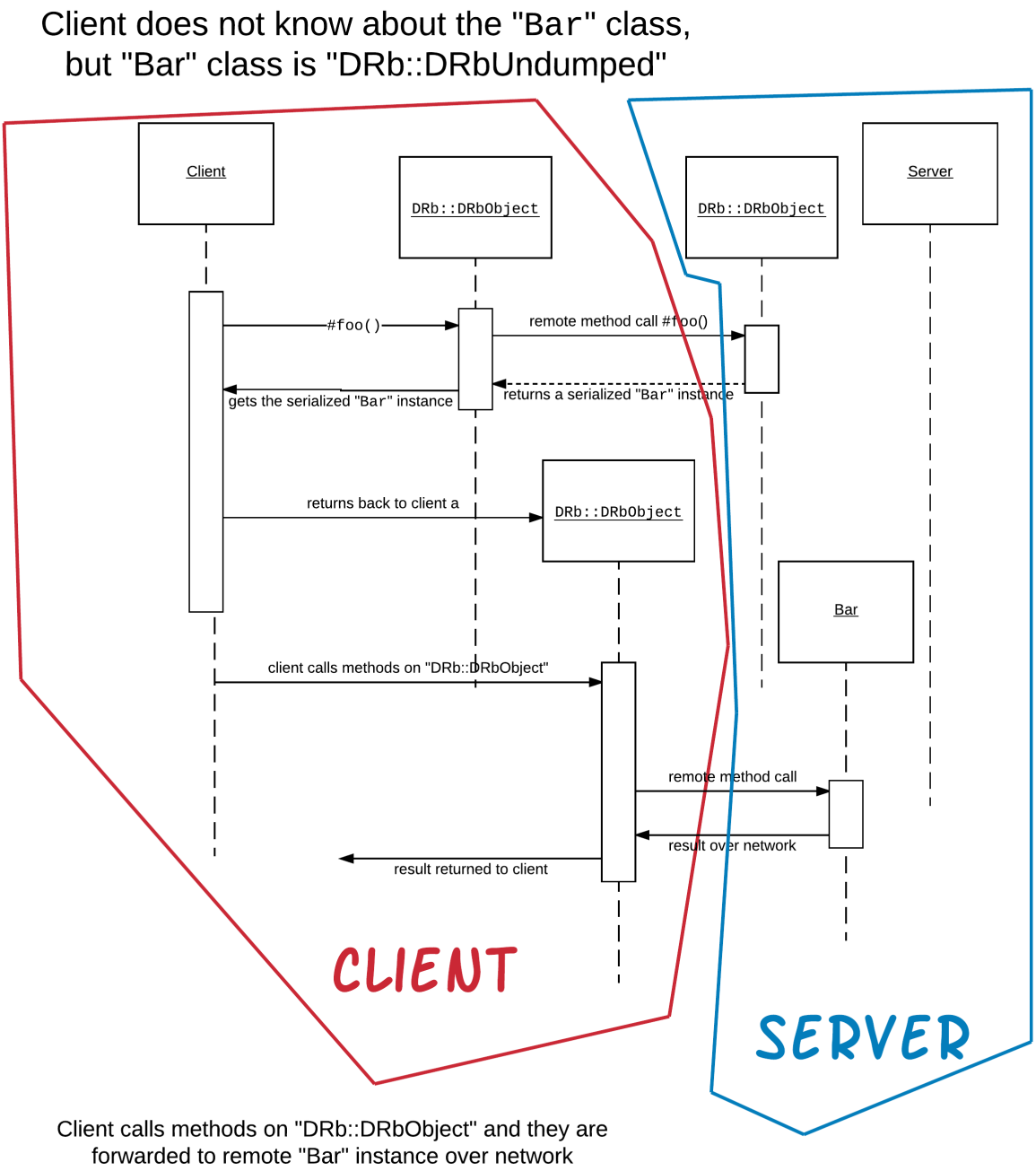
(the above code snippet online)

Now, start your client and while it is sleeping for 10 seconds, kill your server. You will not see your client failing.

Keeping Only a Reference To Remote Object

Do we have to let client know of the class of the remote object returned? No, we don't. But if we don't, we get a `DRb::DRbUnknown` instance and we cannot actually call methods on this returned object. Or, at least, this is not easy. So, how can we avoid defining the class details of the returned object in the client code, but still being able to call methods on the remote object?

This is only possible if the remote object class allows it. If the class includes `DRb::DRbUndumped` module then client does not get a `DRb::DRbUnknown` instance, but it gets a `DRb::DRbObject` instance and it can call any method on this object. But the method, in that case, it will be called in the server process.



Client Can Now Call Remote Methods Which are Executed At the Server Side

Let's see that. Take the `my_time.rb` code and put it back to the `server.rb` file. Then delete the `my_time.rb` file completely. We are going to have this class defined in the `server.rb` file only:

```
1  # File: drb-example1/server.rb
2  #
3  require 'drb/drb'
4
5  URI = "druby://localhost:8787"
6
7  class MyTime
8    include DRb::DRbUndumped
9
10   def initialize(a_time_instance)
11     @a_time_instance = a_time_instance
12   end
13
14   def to_s
15     @a_time_instance.strftime("%Y%m%d%H%M%S")
16   end
17 end
18
19 class TimeServer
20   def get_current_time
21     MyTime.new(Time.now)
22   end
23 end
24
25 SERVER_OBJECT = TimeServer.new
26
27 puts 'Starting server ....that can tell the time...(Hit Ctrl+C to terminate me)'
28 puts "(I am here: #{URI})..."
29
30 DRb.start_service(URI, SERVER_OBJECT)
31
32 # Wait for the drb server thread to finish before exiting.
33 DRb.thread.join
```

(the above code snippet online)

Watch out for line 8. It is a new line that needs to be added. This makes the `MyTime` instance not to travel and be reconstructed at the client side. Only a remote reference will be sent to the client.

Save and start your server.

Your `client.rb` should be like this:

```ruby
# File: drb-example1/client.rb
#
require 'drb/drb'

SERVER_URI = "druby://localhost:8787"

puts 'I am a client and I want to know the current time...'
puts "...I am going to ask #{SERVER_URI}"

timeserver = DRbObject.new_with_uri(SERVER_URI)

timeserver_result = timeserver.get_current_time

puts "...got back this: #{timeserver_result}"

puts "...timeserver is #{timeserver.inspect}"
puts "...timeserver_result class is #{timeserver_result.class}"
puts "...timeserver_result inspect is #{timeserver_result.inspect}"
```

(the above code snippet online)

Note that the `client.rb` does not know about the `MyTime` class. Now that `MyTime` class includes `DRb::DRbUndumped`, how will the `client.rb` behave? Let's run it:

```
drb-example1 > $ ruby client.rb
I am a client and I want to know the current time...
...I am going to ask druby://localhost:8787
...got back this: 20170424223727
...timeserver is #<DRb::DRbObject:0x007fa1e5003460 @uri="druby://localhost:8787", \
 @ref=nil>
...timeserver_result class is DRb::DRbObject
...timeserver_result inspect is #<DRb::DRbObject:0x007fa1e5002560 @uri="druby://l\
ocalhost:8787", @ref=70304802116320>
drb-example1 > $
```

(the above code snippet online)

Perfect. The `...got back this: 20170424223727` confirms that the method `#to_s` has been called without problem. Also, do you see the class of the `timeserver_result`? It is an instance of `DRb::DRbObject`. This makes it possible to call its methods at the server process.

To confirm that the `#to_s` is now executed at the server side process, let's put the `sleep 10` again and while the `client.rb` sleeps, we will kill the server.

```ruby
# File: drb-example1/client.rb
#
require 'drb/drb'

SERVER_URI = "druby://localhost:8787"

puts 'I am a client and I want to know the current time...'
puts "...I am going to ask #{SERVER_URI}"

timeserver = DRbObject.new_with_uri(SERVER_URI)

timeserver_result = timeserver.get_current_time

puts "Sleeping for 10 seconds to give enough time for the server to be killed..."
sleep 10

puts "...got back this: #{timeserver_result}"

puts "...timeserver is #{timeserver.inspect}"
puts "...timeserver_result class is #{timeserver_result.class}"
puts "...timeserver_result inspect is #{timeserver_result.inspect}"
```

(the above code snippet online)

Now run the `client.rb` and kill the server while client is sleeping. This is what you will get:

```
drb-example1 > $ ruby client.rb
I am a client and I want to know the current time...
...I am going to ask druby://localhost:8787
Sleeping for 10 seconds to give enough time for the server to be killed...
.rvm/rubies/ruby-2.2.3/lib/ruby/2.2.0/drb/drb.rb:744:in `rescue in block in open'\
: druby://localhost:8787 - #<Errno::ECONNREFUSED: Connection refused - connect(2)\
 for "localhost" port 8787> (DRb::DRbConnError)
        from ....rvm/rubies/ruby-2.2.3/lib/ruby/2.2.0/drb/drb.rb:738:in `block in\
 open'
        from ....rvm/rubies/ruby-2.2.3/lib/ruby/2.2.0/drb/drb.rb:737:in `each'
        from ....rvm/rubies/ruby-2.2.3/lib/ruby/2.2.0/drb/drb.rb:737:in `open'
        from ....rvm/rubies/ruby-2.2.3/lib/ruby/2.2.0/drb/drb.rb:1248:in `initial\
ize'
        from ....rvm/rubies/ruby-2.2.3/lib/ruby/2.2.0/drb/drb.rb:1228:in `new'
        from ....rvm/rubies/ruby-2.2.3/lib/ruby/2.2.0/drb/drb.rb:1228:in `open'
        from ....rvm/rubies/ruby-2.2.3/lib/ruby/2.2.0/drb/drb.rb:1141:in `block i\
n method_missing'
        from ....rvm/rubies/ruby-2.2.3/lib/ruby/2.2.0/drb/drb.rb:1160:in `with_fr\
iend'
        from ....rvm/rubies/ruby-2.2.3/lib/ruby/2.2.0/drb/drb.rb:1140:in `method_\
missing'
```

```
22          from client.rb:17:in `<main>'
23 drb-example1 > $
```

(the above code snippet online)

Boom! The `#to_s` in this case failed. Because we now don't have a local representation of the object and the execution takes place at the server side.

In summary, when you design your classes that are going to be travelling from server to client, then make sure that you know which classes will be including `DRb::DRbUndumped` and which ones not. Which objects need to travel to the client side and their class needs to be known to the client vs the objects that will not have to travel to the client and only a remote reference would have been enough.

Missing Method Is Sent To Remote Object

Let's now proceed to a new example that will demonstrate the following:

When the client code calls a method on a DRb object that does not exist on this object, i.e. the `DRb::DRbObject` does not respond to this method, then this method call is forwarded to the remote object

The reason this is of great importance is going to be revealed later on, in the discussion about Security. Until then, let's proceed with the example.

Start the `drb-example2` project and write the server code as follows:

```ruby
# File: drb-example2/server.rb
#
require 'drb/drb'

class MyRemoteClass
  def foo
    "foo #{object_id}"
  end

  def inspect
    "my remote classssssss #{object_id}"
  end
end

my_remote_class = MyRemoteClass.new

DRb.start_service('druby://localhost:8787', my_remote_class)

DRb.thread.join
```

(the above code snippet online)

This is a very simple DRb server. It exposes the method `#inspect` of the object instances of class `MyRemoteClass`.

Now, let's write the client side:

```
# File: drb-example2/client.rb
#
require 'drb/drb'

ro = DRbObject.new_with_uri('druby://localhost:8787')

puts ro.foo
puts ro.inspect
```

(the above code snippet online)

This is even simpler. We just call `#foo` and then `#inspect` on the remote object. Let's start the server and then run the client on another terminal. This is what you will get:

```
drb-example2 > $ ruby client.rb
foo 70296627378820
#<DRb::DRbObject:0x007fd5ab8bf618 @uri="druby://localhost:8787", @ref=nil>
drb-example2 > $
```

(the above code snippet online)

Right! The method `#foo` was actually called on the server side process, whereas the method `#inspect` was called on the client side process. Although we thought that we called `#inspect` on the remote object, actually, we called the `#inspect` on the local representative of the remote object, i.e. on the `DRb::DRbObject` instance. That's why we got back `#<DRb::DRbObject:0x007fc83a987650 @uri="dr...` and not something like `my remote classssssss....`

Whenever we call a method on the `DRb::DRbObject` instance that exists as method on the instance itself, then that method stays here and does not travel to the server process for execution. On the other hand, when we call a method that does not exist in the `DRb::DRbObject` instance methods, then this method is sent over to the server side process for execution, i.e. it is a remote method invocation.

But how can I call the `#inspect` method on the remote object then? You can do that by removing the `#inspect` definition from the `DRb::DRbObject` instance, before actually calling that on this instance. Having removed that, the method invocation will be forwarded to the remote object.

Let's do that. This is the new version of the client that finally manages to call the `#inspect` on the remote object:

```
1  # File: drb-example2/client.rb
2  #
3  require 'drb/drb'
4
5  ro = DRbObject.new_with_uri('druby://localhost:8787')
6
7  puts ro.foo
8
9  module DRb
10   class DRbObject
11     undef_method :inspect
12   end
13 end
14
15 puts ro.inspect
```

(the above code snippet online)

In between lines 9 and 11, we undefine the method `#inspect` from the instances of the class `DRb::DRbObject`.

Let's now run the client again:

```
1  foo 70296627378820
2  my remote classssss 70296627378820
```

(the above code snippet online)

Nice! The `#inspect` has been called on the remote instance.

A better alternative, though, would have been to have undefined the method `#inspect` only for the `ro` instance and not from all `DRb::DRbObject` objects, like we did. This version of client code below, removes the `#inspect` definition only from the particular `ro` instance.

```
1  # File: drb-example2/client.rb
2  #
3  require 'drb/drb'
4
5  ro = DRbObject.new_with_uri('druby://localhost:8787')
6
7  puts ro.foo
8
9  class << ro
10   undef_method :inspect
11 end
12
13 puts ro.inspect
```

(the above code snippet online)

Running client code again, you will get the same result:

```
drb-example2 > $ ruby client.rb
foo 70296627378820
my remote classssssss 70296627378820
drb-example2 > $
```

(the above code snippet online)

Cool! But `#inspect` is only one of the many methods a `DRb::DRbObject` responds to. The following paragraph explains why the replacement technique that we used above (undefinition of the local method definition in order to replace its execution with the one defined at the remote server object) might be proven very dangerous.

Security Concerns

Since a distributed Ruby server allows a client to call a method on the remote/server object, this means that, pottentially, a Ruby client can execute any piece of code at the server side process and host. How this can be done?

Here is a client that invokes a shell command at the server side:

```
1  # File: drb-example2/client.rb
2  #
3  require 'drb/drb'
4
5  ro = DRbObject.new_with_uri('druby://localhost:8787')
6
7  puts ro.foo
8
9  class << ro
10   undef_method :instance_eval
11 end
12
13 ruby_code =<<RUBY_CODE
14 `rm server.rb`
15 RUBY_CODE
16
17 ro.instance_eval ruby_code
```

(the above code snippet online)

This client code, using the technique of undefining a method that we saw earlier, it undefines the method `#instance_eval`. Having done that, forces DRb to actually forward the method call to the remote object. This means that the argument given in this method invocation on line 17 will be used as argument to execute arbitrary Ruby code at the **server** process and host machine.

In other words, the above client code will **remove the `server.rb` file from the directory where server is running**.

Pretty scary, isn't it? Imagine what would have happened if the command was `rm -f -R .` instead.

Now that we know how dangerous DRb might be we need to learn whether we can increase the security level.

Let's see how:

Tainted Variables

In Ruby all the strings that are coming from the outside world are marked as *tainted*. Try the following Ruby program:

```
1  # File: tainted_example.rb
2  #
3  external_string = ARGV[0]
4  internal_string = 'Hello World'
5
6  puts "External string tainted?: #{external_string.tainted?}"
7  puts "Internal string tainted:: #{internal_string.tainted?}"
```

(the above code snippet online)

This program prints the *tainted?* flag for two strings. One that is coming from the outside world and stored in the variable `external_string` and another one that takes its value from a local literal string, stored in the variable `internal_string`.

Run this program as follows:

```
1  $ ruby tainted_example.rb foo
2  External string tainted?: true
3  Internal string tainted:: false
4  $
```

(the above code snippet online)

You will see that the external string is considered *tainted?* whereas the internal string is considered *untainted?*. What does this have to do with our security subject? It is the *tainted* variables that we need to be careful for and Ruby makes sure that these values follow specific rules and their usage can be constrained. This is explained in the following paragraph:

Ruby Safe Levels

Ruby is running on a specific safe level. The safe level is always stored in the global variable `$SAFE`.

Default Safe Level 0

```
1  # File: default_safe_level.rb
2  #
3  puts $SAFE
```

(the above code snippet online)

Run the above program and you will get the value of the default safe level. It is `0`.

In this level, the *tainted* data, i.e. data that is coming from the outside world are no different to the safe data.

Safe Level >= 1

When the save level is `>= 1`, then Ruby does not allow the use of tainted data from potentially dangerous operations. One such operation is the `eval` or the `instance_eval`.

Let's see that with the DRb server we created earlier. Here is its new version:

```
1  # File: drb-example2/server.rb
2  #
3  require 'drb/drb'
4
5  $SAFE = 1
6
7  class MyRemoteClass
8    def foo
9      "foo #{object_id}"
10   end
11
12   def inspect
13     "my remote classssssss #{object_id}"
14   end
15 end
16
17 my_remote_class = MyRemoteClass.new
18
19 DRb.start_service('druby://localhost:8787', my_remote_class)
20
21 DRb.thread.join
```

(the above code snippet online)

The only difference here is the `$SAFE = 1` statement on line 5.

Now start the server and then try to run the dangerous client again (the one that undefined `instance_eval` and executed `rm server.rb`). Here is what you will get:

```
drb-example2 > $ ruby client.rb
foo 70127315958480
/.../drb/drb.rb:578:in `load': connection closed (DRb::DRbConnError)
        from ...2.2.3/lib/ruby/2.2.0/drb/drb.rb:640:in `recv_reply'
        from ...2.2.3/lib/ruby/2.2.0/drb/drb.rb:940:in `recv_reply'
        from ...2.2.3/lib/ruby/2.2.0/drb/drb.rb:1254:in `send_message'
        from ...2.2.3/lib/ruby/2.2.0/drb/drb.rb:1142:in `block (2 levels) in meth\
od_missing'
        from ...2.2.3/lib/ruby/2.2.0/drb/drb.rb:1229:in `open'
        from ...2.2.3/lib/ruby/2.2.0/drb/drb.rb:1141:in `block in method_missing'
        from ...2.2.3/lib/ruby/2.2.0/drb/drb.rb:1160:in `with_friend'
        from ...2.2.3/lib/ruby/2.2.0/drb/drb.rb:1140:in `method_missing'
        from client.rb:17:in `<main>'
drb-example2 > $
```

(the above code snippet online)

Line `puts ro.foo` in the client code was successfully executed. But then, the `ro.instance_eval ruby_code` was not. Server was very harsh with this client and closed the connection, refusing to execute its dangerous code. Note that the file `server.rb` is still there and was not deleted like it did before, when we had the unsafe version of the server code.

Other Safe Levels

The `$SAFE` variable can be set to other values too:

1. $SAFE >= 2: This prohibits Ruby interpreter from loading (`require` or `load`) a Ruby program from a globally writable location.
2. $SAFE >= 3: This level renders any newly created object as *tainted*, hence, pottentially unsafe.
3. $SAFE >= 4: In this level Ruby partitions the running program in two. Nontainted objects, i.e. objects created from the program itself cannot be modified.

The explanation of these safe levels is out of the scope of this course.

Closing Note

We have seen how we can create Ruby programs that are distributed to many processes deployed on different machines. They will be able to communicate via remote method invocations.

Tasks and Quizzes

> **Before you continue, you may want to know that**: You can sign up to Tech Career Booster and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

Task A Details

You need to use the knowledge that you have acquired in this chapter in order to build a Ruby program that is split in two. A client and a server. The server needs to expose a front object that would work as a factory for other objects. The front object, would respond to a single method `#build(<object to build>)`. This method would take as input one of the following string values:

1. `encryptor`
2. `base64`
3. `compressor`

It would return back to the client an object that would be able to carry out some basic string operation.

1. If `encryptor` is given, the client would be given back an object at hand that would respond to the method `#encrypt(<string to encrypt>)`. This method would return the input string to an encrypted form. The encryption can be anything that you like. Also, the same object would respond to the method `#decrypt(<string to decrypt>)`. This method would return back the clear string of the encrypted version given as input argument.
2. If `base64` is given, the client should then have an object at hand that would respond to the methods `#encode(<string to encode>)` and `#decode(<string to decode>)`. The first method would return back the string given but in its Base64 strict encoded version. The second method would return back the string given but in its Base64 strict decoded version (see here for a help).
3. If `compressor` is given, the client gets back an object that exposes two methods again. The first method is `#compress(<string to compress>)` and returns back a compressed version of the input string given. The second method is `#uncompress(<string to uncompress>)`, which takes as input a compressed version of a string and returns back is uncompressed version. You can choose any library to compress/uncompress, but we are suggesting you using ZLib.

Below, you can see an indicative run of the client using these server objects:

```
task-a > $ ruby client.rb 'I Love Computer Programming with Tech Career Booster. \
Great content and fantastic Mentors'
Initial string is: I Love Computer Programming with Tech Career Booster. Great co\
ntent and fantastic Mentors
Encrypted: J!Mpwf!Dpnqvufs!Qsphsbnnjoh!xjui!Ufdi!Dbsffs!Cpptufs/!Hsfbu!dpoufou!bo\
e!gboubtujd!Nfoupst
Decrypted: I Love Computer Programming with Tech Career Booster. Great content an\
d fantastic Mentors
Base64 encoded: SSBMb3ZlIENvbXB1dGVyIFByb2dyYW1taW5nIHdpdGggVGVjaCBDYXJlZXIgQm9vc\
3Rlci4gR3JlYXQgY29udGVudCBhbmQgZmFudGFzdGljIE1lbnRvcnM=
Base64 decoded: I Love Computer Programming with Tech Career Booster. Great conte\
nt and fantastic Mentors
Compressed Size: 87
```

```
14  Uncompressed Size: 89
15  task-a > $
```

Note that the string to work on, is given as input run-time argument on the command line.

In this version of the code, you need make sure that client code is becoming aware of the classes of the different objects and the execution of the string manipulation methods take place in the client memory and process space.

Important: Upload your work on your Github account so that you mentor can evaluate it online.

Task B - Details

This should the same Ruby application like the one in Task A, but the client does not need to be aware of the classes that implement the different objects. The execution of the string manipulation methods should take place in the server memory and process space.

Important: Upload your work on your Github account so that you mentor can evaluate it onlin

2 - Introduction to TCP/IP Sockets

Summary

TCP IP Sockets

In the world of Internet, the majority of the devices communicate using TCP/IP sockets. A client sends a message to a server and gets back a response, all with the help of TCP/IP sockets. This chapter is an introduction to this old technology that has survived the test of time and that is very useful for people to be connected over Internet. Learning about sockets will prepare you to later on study the HTTP protocol which sits on top of them.

Learning Goals

1. Learn about TCP/IP sockets.
2. Learn about Berkeley Sockets API.
3. Learn about the TCP connections.
4. Learn about the server TCP socket lifecycle.
5. Learn about the `netcat` program.
6. Learn about the high-level Ruby functions that help us write client and server sockets programs.
7. Learn about IPv4 and IPv6 addressing.
8. Learn how you can create pair of sockets to support both versions of IP addressing.
9. Learn about the client TCP socket lifecycle.
10. Learn how to create a dialogue between a client and a server.
11. Learn about the EOF character.
12. Learn how to limit the size of data read from a connection.
13. Learn about the partial reads.

Introduction

TCP/IP (Transmission Control Protocol/Internet Protocol) sockets is a set of programming API that allows us to write Ruby applications deployed on the Internet. It is the Berkeley Sockets API that we will rely on in this course. This API has been invented back in 1983 and has truly stood the test of time.

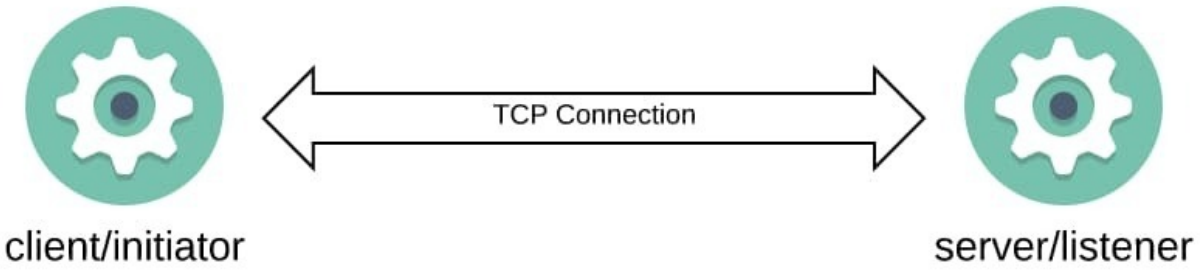
Although we will internally be using the Berkeley Sockets API, this will be done via the Ruby libraries that wrap the calls to this API. Hence, you will have the chance to learn details about

the Berkeley Sockets API that are independent of the programming language that you use, but, on top of that, we will show you how Ruby has made it much easier to use this API via their libraries.

Having said the above, let's start with the Ruby Sockets API.

TCP Connections

A TCP Connection involves two endpoints. The client, which is also called *initiator* and the server, which is also called *listener*.



TCP Connection End Points

Server TCP Socket LifeCycle

A server TCP socket lifecycle includes the following:

1. create the TCP socket
2. bind to an IP/PORT address
3. listen for incoming connections from a client
4. accept an incoming connection
5. close an incoming connection and the socket

The following is a program that demonstrates the above:

```
1  # File: sockets-server-1.rb
2  #
3  require 'socket'
4
5  # 1. create the socket
6  socket = Socket.new(:INET, :STREAM)
7
8  # 2. Bind:
9  #
10 # 2.1. Create a structure to hold the address for listening
11 address = Socket.pack_sockaddr_in(8080, '0.0.0.0')
12 # 2.2. Bind to this address
13 socket.bind(address)
```

```
14
15 # 3. Listen for incoming connections:
16 socket.listen(5) # 5 is the size of the pending connections queue
17
18 # 4. Accept an incoming connection:
19 connection, remote_address = socket.accept
20
21 puts "Connection: #{connection.inspect}"
22 puts "Remote address: #{remote_address.inspect}"
```

(the above code snippet online)

These are the things that you need to be aware of about the above program:

1. It requires the `socket` file. This is because sockets in Ruby are part of the standard library and, hence, they are not required by default.
2. On line 6 we take the first step to create the socket object. The `:INET` and `:STREAM` parameters are there in order to create an Internet v4 TCP socket.
3. Then we just call `socket.listen(...)` on line 16, which is the third step in the server socket lifecycle. The `#listen` method takes as argument the size of the *pending connections queue*. When a client tries to connect to a server and the server is busy, in the middle of processing the previous connection, then latest incoming connection will be first put in the *pending connections queue* until it is accepted by the server. You cannot set this number to a very big number for various reasons. One is that you don't want your clients to wait for connections for too long. If it appears that you need bigger queue, this might mean that your server does not process connections quickly, something that you would better remedy. Also, the queue consumes system resources which are shared with other processes in your host machine. Nevertheless, the maximum number that you can use here is `Socket::SOMAXCONN` which on my machine was `128`. Also, you don't want this number to be very small, because if you do, you increase the chances your clients get a connection refused error. For this example, we set the number to `5`, but feel free to experiment.
4. This is the 4th step in the server socket lifecycle. When we call the `#accept` on a server socket, then the process thread calling this, it blocks until a new client connection comes in. When the client connection finally comes in, then this method returns two elements. A connection object and a remote address object. The first has the details of the connection established and the second has specific details about the address of the remote client that the connection was established with.

Let's run this program on a terminal:

```
1 $ ruby sockets-server-1.rb
```

(the above code snippet online)

When you start this program, the process does not terminate. It is the line 19, `socket.accept` that blocks the server process until a client comes in.

Can we have a client connect to this socket? One program that you can have handy for such things is `netcat` or `nc` in Linux. Try the following on *another* terminal:

```
$ nc localhost 8080
$
```

(the above code snippet online)

You will see that the server process will execute lines 21 and 22 and will terminate:

```
Connection: #<Socket:fd 8>
Remote address: #<Addrinfo: 127.0.0.1:57197 TCP>
$
```

(the above code snippet online)

What has just happened is that the `nc` program worked as a client and connected to the port `8080` on the local machine, i.e. on the machine that the server process (`sockets-server-1.rb`) was waiting for an incoming connection. So, the server process was unblocked from the `socket.accept` call and it executed the two lines that printed the connection and the remote address information.

Accept Loop

We saw that the server program that we wrote above terminated when the client connected to it. Usually, you wouldn't like to do that. You would like to be able to accept more connections, after accepting the first.

You can easily do that with a `loop` around `accept`. Here is the new version of our server:

```ruby
# File: sockets-server-2.rb
#
require 'socket'

socket = Socket.new(:INET, :STREAM)

address = Socket.pack_sockaddr_in(8080, '0.0.0.0')
socket.bind(address)

socket.listen(5)

loop do
  connection, _ = socket.accept
  puts "Server local address: #{connection.local_address.inspect}"
  puts "Client remote address: #{connection.remote_address.inspect}"
  connection.close
end
```

(the above code snippet online)

You can see that we have wrapped the `socket.accept` into a `loop do ... end` block. Which means that this call after accepting an incoming client request, it prints the details of the connection and then closes the connection before going to accept a new one.

Hint: When we don't care about working on the return value of a Ruby method call, we can assign it to _.

Note also that we print the details of the local and remote address as they are returned from the `connection` object.

Here is an example of what we see if we start this new version of the server and we use another terminal to call `nc localhost 8080`.

```
$ ruby sockets-server-2.rb
Server local address: #<Addrinfo: 127.0.0.1:8080 TCP>
Client remote address: #<Addrinfo: 127.0.0.1:57370 TCP>
```

(the above code snippet online)

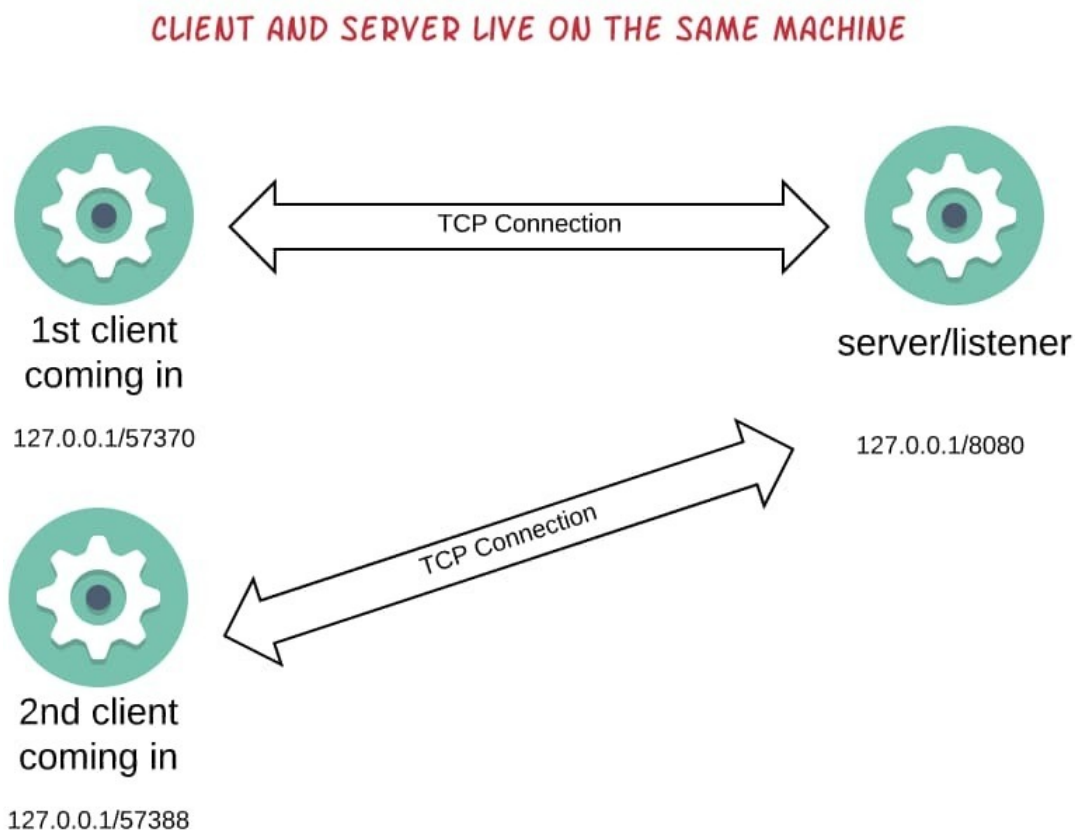
and the server stays there waiting for more connections. Try once more to call `nc localhost 8080`. This is what you will be looking at, at the server terminal:

```
$ ruby sockets-server-2.rb
Server local address: #<Addrinfo: 127.0.0.1:8080 TCP>
Client remote address: #<Addrinfo: 127.0.0.1:57370 TCP>
Server local address: #<Addrinfo: 127.0.0.1:8080 TCP>
Client remote address: #<Addrinfo: 127.0.0.1:57388 TCP>
```

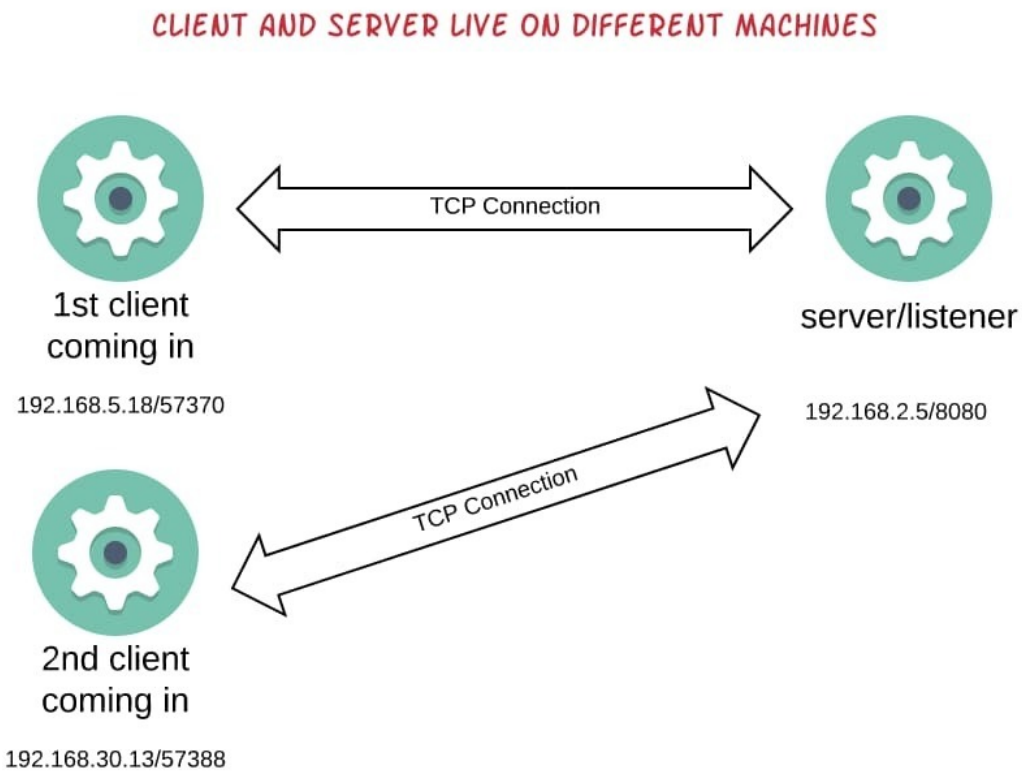
(the above code snippet online)

You can see above that one more pair of local and remote addresses has been printed.

Each TCP connection is a pair of two endpoints as we said above. The server point and the client point. They are both represented with the `Addrinfo` object. Two properties of this object are more important here. The IP and the port. You can see that the server port printed (8080) is the one that we bound the server on. On the other hand, the client port is always ephemeral and assigned on the fly on each client coming in.



Note that client and server do not have to be living in the same host. In the picture above we see the client living in the same machine like the server does, both having same IP, `127.0.0.1`. But in the following, you see an example with client and server living on completely different machines.



Note that when a connection is not needed any more by the server side, it is `closed`. This is necessary because it releases invaluable system resources.

Information: Sometimes, you may try to start your server and you might get the following error `bash sockets-server-2.rb:8:in `bind': Address already in use - bind(2) for 0.0.0.0:8080 (Errno::EADDRINUSE) from sockets-server-2.rb:8:in `<main>'` (the above code snippet online) This means that your server is trying to bind to port that is already in use. Make sure that you don't have another server process running at the same time (and bound to the same port). Also, sometimes, the port is not made available by the system, immediately after the server terminates. In that case, you will have to wait for a minute or so, before trying to bring the server up again on the same port. Or you can just start your server on another port.

Using Ruby API

We have seen how the server can use the Ruby version of the Berkeley Sockets API. However, Ruby takes it one step further and offers us an API that makes creating, binding and listening on sockets much easier. Let's see that.

Write the following new version of the server process (`sockets-server-3.rb`):

```ruby
# File: sockets-server-3.rb
#
require 'socket'

server = TCPServer.new(8080)

loop do
  connection = server.accept
  puts "Server local address: #{connection.local_address.inspect}"
  puts "Client remote address: #{connection.remote_address.inspect}"
  connection.close
end
```

(the above code snippet online)

Then, start the server by running `ruby sockets-server-3.rb`. On another terminal, issue `nc localhost 8080`. Do it twice. You will see similar output printed at the server terminal like you did with the version `sockets-server-2.rb`. Something like this:

```
$ ruby sockets-server-3.rb
Server local address: #<Addrinfo: 127.0.0.1:8080 TCP>
Client remote address: #<Addrinfo: 127.0.0.1:61369 TCP>
Server local address: #<Addrinfo: 127.0.0.1:8080 TCP>
Client remote address: #<Addrinfo: 127.0.0.1:61371 TCP>
```

(the above code snippet online)

`sockets-server-3.rb` functions exactly the same like the `sockets-server-2.rb`. Instead of calling 4 lines of code like this:

```ruby
socket = Socket.new(:INET, :STREAM)

address = Socket.pack_sockaddr_in(8080, '0.0.0.0')
socket.bind(address)

socket.listen(5)
```

(the above code snippet online)

we just call one line of code like this:

```ruby
server = TCPServer.new(8080)
```

(the above code snippet online)

Which one would you prefer? I definitely prefer the latter one.

Note, however, one more difference. The `TCPServer.new` returns a `TCPServer` object and not a socket object. So, calling `server.accept` on line 8, then it returns only the `connection` and not both the `connection` and the `remote_address`.

IPv4 VS IPv6

The most common IP addresses are addresses like this: `46.176.104.17` i.e. addresses with 4 integer numbers, each one starting from 0 up to 255. These are the v4 IP addresses. However, since more and more computers are acquiring IP addresses every day, which means that IP v4 addressing scheme is running out of addresses, a new version of IP addressing scheme has been invented, that would cover for many more IP addresses. This new scheme is called v6 and this is an example of an IP v6 address: `0:0:0:0:0:ffff:2eb0:6811`.

Ruby allows you to bind a server to a socket that complies to both IPv4 and IPv6. In fact, we are talking about two sockets that could be *listened* together and return back one connection when either of them is contacted by a client.

The Ruby API that creates the two sockets is `Socket.tcp_server_sockets`. Then you can use the `Socket.accept_loop(sockets)` to wait for any client at either of these two sockets.

Here is an example that uses this technique (file: `sockets-server-4.rb`):

```
1  # File: sockets-server-4.rb
2  #
3  require 'socket'
4
5  sockets = Socket.tcp_server_sockets(8080)
6  puts "sockets: #{sockets.inspect}"
7
8  Socket.accept_loop(sockets) do |connection|
9    puts "Server local address: #{connection.local_address.inspect}"
10   puts "Client remote address: #{connection.remote_address.inspect}"
11   connection.close
12 end
```

(the above code snippet online)

On line 5, we return two sockets in the array `sockets`. With the `Socket.accept_loop(sockets) do |connection|` we make sure that we have a server accepting connections to both sockets at the same time, depending on how the client wants to connect to the server, i.e. either using IPv4 or IPv6 addressing scheme. In any case, the new connection created is given in the block level variable. Inside the block, server is handling the connection. Before ending the processing, it closes the connection like before.

Try to run the server program. You will initially see this:

```
1  $ ruby sockets-server-4.rb
2  sockets: [#<Socket:fd 8>, #<Socket:fd 9>]
```

(the above code snippet online)

The server waits for a client to connect. Now run the command `nc localhost 8080` on another terminal. You will get something like this at the server side:

```
Server local address: #<Addrinfo: [::1]:8080 TCP>
Client remote address: #<Addrinfo: [::1]:61589 TCP>
```

(the above code snippet online)

Then try the localhost/127.0.0.1 representation in IPv6 version: `nc 0:0:0:0:0:ffff:7f00:1 8080`. You will get something like this:

```
Server local address: #<Addrinfo: 127.0.0.1:8080 TCP>
Client remote address: #<Addrinfo: 127.0.0.1:61605 TCP>
```

(the above code snippet online)

In the previous examples, you can see how server is representing either the IPv4 or IPv6 addresses (`AddrInfo` instances).

Ruby API Is Even Easier

But Ruby API for TCP sockets that support both IPv4 and IPv6 and loop while accepting connections on a specific port is even easier. See the version `sockets-server-5.rb`:

```ruby
# File: sockets-server-5.rb
#
require 'socket'

Socket.tcp_server_loop(8080) do |connection|
  puts "Server local address: #{connection.local_address.inspect}"
  puts "Client remote address: #{connection.remote_address.inspect}"
  connection.close
end
```

(the above code snippet online)

Believe it or not (you only have to run it and convince yourself), this server program is exactly the same like the one in `sockets-server-4.rb`. But with much less lines of code.

Client Program

We have talked about the server side of the TCP socket. Let's see now the details on how we build a client that would connect to a server socket.

The version `sockets-client-1.rb` uses the low level API. We will see this first before we use the high-level Ruby API.

```
1  # File: sockets-client-1.rb
2  #
3  require 'socket'
4
5  socket = Socket.new(:INET, :STREAM)
6  remote_addr = Socket.pack_sockaddr_in(8080, '127.0.0.1')
7  socket.connect(remote_addr)
8  socket.close
```

(the above code snippet online)

On line 5, we first create the socket. We define it to be an IPv4 TCP/IP socket. Then on line 6 we convert the address that we want to connect to into a structure that is necessary for remote addressing. We specify also that we want to connect to port 8080 of the localhost. Then on line 7, we connect using the `socket.connect(remote_addr)`, which basically tells the client to initiate a connection to the server running on localhost/8080.

We finally close the socket to release any resources.

Having the `sockets-server-5.rb` programming running on a terminal, run the above client program on another:

```
1  $ ruby sockets-client-1.rb
2  $
```

(the above code snippet online)

You will not see anything on clients terminal. The client will terminate immediately. What you will see on the server terminal will be something like this (something that we have already seen earlier):

```
1  $ ruby sockets-server-5.rb
2  Server local address: #<Addrinfo: 127.0.0.1:8080 TCP>
3  Client remote address: #<Addrinfo: 127.0.0.1:61893 TCP>
4
5  $
```

(the above code snippet online)

which proves that the client has been successfully connected to the server.

Client Using the Ruby API.

Ruby API offers a higher level interface to connect to a client using TCP, both for IPv4 and IPv6. Here is the version of the previous client using this higher level API.

```ruby
# File: sockets-client-2.rb
#
require 'socket'

Socket.tcp('127.0.0.1', 8080) do |connection|
  connection.close
end
```

(the above code snippet online)

Pretty neat. Isn't it? Try this client now (while having the `sockets-server-5.rb` still running). You will see something like this at the server terminal:

```
Server local address: #<Addrinfo: 127.0.0.1:8080 TCP>
Client remote address: #<Addrinfo: 127.0.0.1:61928 TCP>
```

(the above code snippet online)

Exchange Data

The examples so far didn't do something useful. Useful things start to happen when we have the client request data, ask questions, and the server return back responses.

Server Getting The Question

Let's enhance our latest server example to be able to read the question, i.e. get data from the client.

```ruby
# File: server-reading-from-connection.rb
#
require 'socket'

Socket.tcp_server_loop(8080) do |connection|
  puts 'starting to read ...'
  puts connection.read
  puts '...end reading'

  puts 'closing connection...'
  connection.close
  puts '...connection closed'
end

puts 'Server terminates'
```

(the above code snippet online)

This server version is similar to the previous one, but it also has the line 7, `connection.read` that renders server into a *reading* mode. In other words, server reads whatever is put on the connection. When it finishes reading (we will see what does this exactly mean) we just print some extra messages.

Let's run the server on a terminal (`ruby server-reading-from-connection.rb`) and then use the `echo 'hello world' | nc localhost 8080` command on another terminal. This `netcat` command is going to send the string `hello world` to the server waiting for data on port `8080`.

This is what we will see at the server side after we send the `hello world` string with the above command.

```
$ ruby server-reading-from-connection.rb
starting to read ...
hello world
...end reading
closing connection...
...connection closed
```

(the above code snippet online)

Cool! `connection.read` was executed. The string `hello word` was printed at the server side, thanks to the command `puts connection.read`. In other words, the `connection.read` was executed and the returned value, which was the string sent by the client, was printed on the server side console window.

Server is still running, so you can try more `echo <a string> | nc localhost 8080` commands. You will see the echoed strings being printed at the server side.

A Client That Sends Data

We have used `nc` to send data to our server. Can we write a client that does that?

```
# File: client-writing-to-connection.rb
#
require 'socket'

Socket.tcp('127.0.0.1', 8080) do |connection|
  connection.write ARGV[0]
  connection.close
end
```

(the above code snippet online)

I believe that this must have been expected by you. We use the `connection.write` to write data to the connection. The above client sends to the server the string given as run-time argument on the command line.

If you run this command `ruby client-writing-to-connection.rb 'hello world'` (having server still running), you will see server printing the words:

```
starting to read ...
hello world
...end reading
closing connection...
...connection closed
```

(the above code snippet online) again.

Server Waits for EOF Character

Before we continue it is very important that you understand that server `connection.read` reads data until it encounters the datum which is called EOF (End Of File). This is something like *there is no more data* signal that is sent from the client to the server and makes the `connection.read` stop waiting for more data from the client. If the EOF is not ever sent by the client, unfortunately, `connection.read` will indefinitely be waiting.

Let's see an occurrence of the non-ending reading case. Instead of using the `echo` command (which tells `nc` when the data finishes) we will use the `tail -f` command which never ends and continuously expects to send data to the `nc` pipe.

Issue the following command on a terminal, while you have your server running:

```
tail -f sockets-server-5.rb | nc localhost 8080
```

(the above code snippet online)

You will see that the above never terminates. `tail` waits for more data to be added to file `sockets-server-5.rb` in order to send it to `nc localhost 8080`. (Note that the choice of the file `sockets-server-5.rb` is completely irrelevant, just ask `tail` to `tail` any existing file). `tail` never sends the EOF character to `nc`. This means that `nc` never sends the EOF to the server too. If you see what happens at the server side, you will see this:

```
starting to read ...
```

(the above code snippet online)

Server is blocked on continuous reading waiting to receive more data until it receives the EOF character.

The original client which used the `connection.write` it did sent the EOF character to the server. Same did the `echo 'hello world' | nc localhost 8080` technique. But this one, with the `tail -f` does not ever send the EOF character.

> Now, let's stop both the server and the `tail -f ...` command by using the keyboard to send <kbd>Ctrl + C</kbd>

Limiting Read Length

The problem we demonstrated above implies also that `connection.read` blocks and does not return any data to the server. It accumulates the data received from the client, but never returns anything to the server until it receives the EOF character.

What we can do in order to give some data back to the server code, even if EOF is not received yet, is that we can limit the read length to a specific small number.

This is the version of the server that does not wait endlessly on the `connection.read` command:

```
1  # File: server-reading-small-chunks-of-data.rb
2  #
3  require 'socket'
4
5  Socket.tcp_server_loop(8080) do |connection|
6    while data = connection.read(10)
7      $stdout.print data
8      $stdout.flush
9    end
10
11   $stdout.puts 'closing connection...'
12   connection.close
13   $stdout.puts '...connection closed'
14 end
15
16 $stdout.puts 'Server terminates'
```

(the above code snippet online)

The difference here being that `connection.read` is provided with a number which specifies the amount of data to be read before returning control back to server and the data read into the variable `data`. In our example, we will continuously be fetching data in chunks of 10. Every 10 bytes we get back from client, we will be printing the data and then we will be going back again to read more data, until the EOF is given by the client or the connection is closed by the client.

Start the server with `ruby server-reading-small-chunks-of-data.rb` and then try to use the `tail -f <an existing file>` technique that we used earlier. You will see that although the server is *blocked* in waiting for data to come in, it now has the ability to print the data sent from client in chunks of 10 characters.

The video below shows this interaction on my machine:

Server Reading Data in Chunks of 10

Nevertheless, `connection.read(<amount of data>)` is blocking. Either waiting for the EOF to be sent, or waiting for the specified amount of data to be sent. This can be quite tricky to handle. And may also end you having a deadlock situation: client waiting for the server and server waiting for the client.

Partial Reads

An alternative to `connection#read` is the `connection#readpartial`. This alternative returns, immediately, the available data. You call it by passing the maximum amount of data that you want to receive. But if less data are available, these will be immediately returned.

Let's see the version of the server that uses partial read:

```
1  # File: server-partial-read.rb
2  #
3  require 'socket'
4
5  Socket.tcp_server_loop(8080) do |connection|
6    begin
7      while data = connection.readpartial(10)
8        $stdout.print data
9        $stdout.flush
10     end
11
12   rescue EOFError
13   end
14
15   $stdout.puts 'closing connection...'
16   connection.close
17   $stdout.puts '...connection closed'
18 end
19
20 $stdout.puts 'Server terminates'
```

(the above code snippet online)

The difference to the lazy `#read` is that the `#readpartial` will read as much as it has and then will return back to the server to continue execution. Note also that, to the contrary of how `#read` behaves, it will raise an `EOFError` exception if the client terminates sending data.

The following video shows how much better the `#readpartial` behaves.

Server Uses ReadPartial

Server Responds Back

We have seen how a client can send a request to the server and how the server can write the request data on the console. But how can we have the server respond back to the client?

This is a new version of the server that takes a *first name* from the client and returns back the *last name* corresponding to that last name.

```ruby
# File: server-responds-back-to-client.rb
#
require 'socket'

last_names = {
  'John' => 'Smith',
  'Mary' => 'Foo',
  'Paul' => 'Fox'
}
Socket.tcp_server_loop(8080) do |connection|
  first_name = connection.read
  puts "First name: #{first_name}"
  answer = last_names.fetch(first_name, 'unknown')
  connection.write answer
  connection.close
end

puts 'Server terminates'
```

(the above code snippet online)

This is a simple server that uses the lazy `connection.read` command in order to read a piece of string from the client. Then it uses the same connection to write back the answer. Then closes the connection (which also sends an EOF to the client).

Let's start the server:

```
ruby server-responds-back-to-client.rb
```

(the above code snippet online)

Now, this is a client that asks for the last name of a first name.

```ruby
# File: client-writing-to-connection.rb
#
require 'socket'

Socket.tcp('127.0.0.1', 8080) do |connection|
  connection.write ARGV[0]
  connection.close_write # This will send the EOF

  last_name = connection.read
  puts last_name

  connection.close
end
```

(the above code snippet online)

The tricky point here is the `#close_write` that you see on line 7. This signals that client does not have anything else to write to the server. This sends the EOF signal and server knows that it does not have to wait for anything else. `#close_write` closes the *write* direction of a connection. Does not close the whole connection. The *read* direction is still available for reading. Hence, client reads the last name returned back by the server and then closes the connection.

Having the server running, try calling the client:

```
1 $ ruby client-writing-to-connection.rb John
2 Smith
3 $ ruby client-writing-to-connection.rb Mary
4 Foo
5 $ ruby client-writing-to-connection.rb Dummy
6 unknown
7 $
```

(the above code snippet online)

Closing Note

We have learned the basics of TCP/IP programming. The clients and servers that we have built are quite primitive. They have many issues that need to be addressed until we call them production-ready. For example, the servers need to introduce some kind of concurrency. Otherwise, serving a client prevents from accepting connections and serving another, until the first request has been completely served.

Tasks and Quizzes

> **Before you continue, you may want to know that**: You can sign up to Tech Career Booster and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

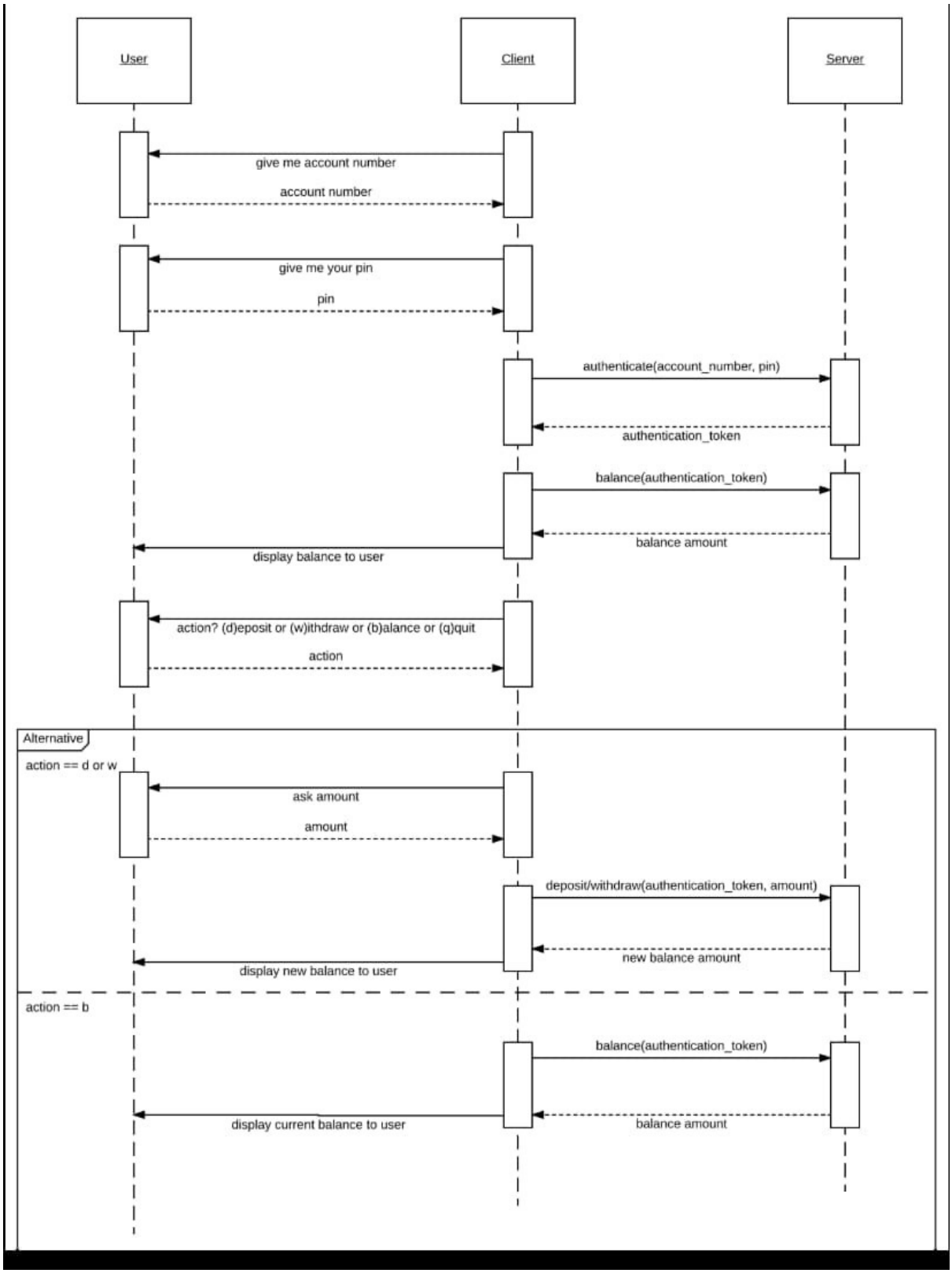
Task Details

You need to use TCP/IP sockets to develop a server that would function as a bank ATM machine. It will have to respond to the following questions:

1. Deposit
2. Withdraw
3. Balance

There is going to be a client that would allow the user to connect to the server and ask for their balance and deposit or withdraw money.

Here is a typical conversation between the client and the server.



Typical Conversation Between Client and Server

1. Initially, user is asked to give their account number and their pin.
2. Then client authenticates user using the `authenticate` service of the server.

3. Then client asks for the current balance of the account, and informs user about it.
4. Then client asks the user to tell it which action they want to carry out.
    1. Deposit
    2. Withdraw
    3. Balance
    4. Quit
5. If deposit or withdraw, client is asking the server to carry out the corresponding action.
6. If balance, client is asking the server to return back the current balance.
7. If quit, then client terminates.

Here is a short video displaying this interaction. In this video we show that we start the server and then we start the client which interacts with the user.

Task - TCP Sockets - ATM Client Server - Typical Interaction

With regards to the user interaction with the client, some things that are not obvious from the video, and that you need to take care of are:

1. If the user does not give correct account number/pin combination, client should terminate by informing user that the credentials were wrong.
2. User selection of an action (deposit, withdraw, balance or quit) should be done with the press of a key without the need to click on Enter. (Hint: The `$stdin.getch` method does that. But you will need to require `io/console` - Google for it).
3. If the server is not running the client should gracefully terminate. (Try to rescue the `Errno::ECONNREFUSED` exception)
4. The server keeps track of the accounts, their pins and their balances using a local *database*. Its database can be either a `yml` or a `JSON` file. Choose whatever you like. Again, the database with the accounts pre-exists.
5. When the server accepts a command to update the balance of an account (deposit or withdraw), then it saves the new balance in its database. So, if you stop the server and you load it again, it will have the new balances read from its database.
6. When a client authenticates, server sends back to the client an authentication token which uniquely identifies the account authenticated for the particular client interaction. Then subsequent calls to update the balance or get the balance use the authentication token returned. Server is able to identify the account from the authentication token and does not need the account number and pin again.

Some hints that will help you implement this project:

1. You will need the lazy `#read` and `#write` commands. Don't look for something more complicated than that.
2. Note that when you want to send the EOF character you can close the write stream of the connection and leave the read stream open in order to read the answer back. Likewise, when you first read and then write, you can close the read stream after you read the message and then use the write stream to write back your answer. In any case, when you no longer need the connection, close it.

3. Your client code needs to format the messages it sends to the server and the server needs to be able to understand the format and take actions according to the content. One way that you can do that is that you can use a `Hash` to encode your message and send over to the server a `JSON` version of the `Hash`. Note that `Hash#to_json` converts a `Hash` to a `JSON` string. Also, `JSON.parse(<a JSON string>)` converts a `JSON` string to a `Hash`. If you go with the `JSON` solution make sure that you require `json` file.
4. You don't have to keep the connection open while the client is running. You can simply create the connection (a different connection) every time you need to send a message to the server.

Try to make your code, client and server, as DRY (Do Not Repeat Yourself) as possible. And as clean as possible. This is very important.

Upload your code to your Github account so that your mentor can evaluate your work