Distributed Internet Applications - DIA

Servers and Sockets
Introduction

The socket API is an Interprocessing Communication (IPC) programming interface originally provided as part of the Berkeley UNIX operating system.

It has been ported to all modern operating systems.

It is a *de facto* standard for programming IPC, and is the basis of more sophisticated IPC interface such as remote procedure call and remote method invocation.
The conceptual model of the socket API
The socket API

A socket API provides a programming construct termed a socket. A process wishing to communicate with another process must create an instance, or instantiate, such a construct.

The two processes then issues operations provided by the API to send and receive data.
Class Hierarchy of Ruby’s Sockets
A socket programming construct can make use of either the UDP or TCP protocol.

Sockets that use UDP for transport are known as **datagram sockets**, while sockets that use TCP are termed **stream sockets**.
UDP Sockets

**UDP sockets** send and receive datagrams.

**UDP is a connectionless service:**

- There isn’t an initial handshaking phase during which a pipe is established between the two processes.

**Datagram sockets** can support both connectionless and connection-oriented communication at the application layer. This is so because even though datagrams are sent or received without the notion of connections at the transport layer, the runtime support of the socket API can create and maintain logical connections for datagrams exchanged between two processes.
Using UDP Sockets: Client

Ask kernel for a socket:

```ruby
s = UDPSocket.new
s = UDPSocket.open
```

Connect to an address:

```ruby
s.connect(host_name, port_num)
```

Send data:

```ruby
s.send(a_string, 0)

s.send(a_string, 0, host_name, port_num)
```

Receive data:

```ruby
s.recvfrom(len)
```
Using UDP Sockets: Server

Ask kernel for a socket:

s = UDPSocket.new

Bind to an address:

s.bind(host_name, port_num)

Send data:

s.send(a_string, 0)

s.send(a_string, 0, host_name, port_num)

Receive data:

s.recvfrom(len)
Connection-oriented & Connectionless UDP Socket

connectionless datagram socket

- a datagram
- a logical connection created and maintained by the runtime support of the datagram socket API

connection-oriented datagram socket
Example: UDP Server

```ruby
# File: recv_udp.rb
require 'socket'
PORT = 10001
HOST = "localhost"

my_socket = UDPSocket.new
my_socket.bind(HOST, PORT)

msg = my_socket.recvfrom(80)
puts "Message from #{msg[1][2]} on port #{msg[1][1]}: '#{msg[0]}"
```

Output:

```
Message from localhost on port 49158: 'Hello, World!'
```
Example: UDP Client (1)

You have two choices when sending data:
1. You can connect to a remote UDP socket, and thereafter send data to that port (see this slide)
2. You can specify a host and port for use with every datagram you send (see the next slide)

```ruby
# File: send_udp.rb
require 'socket'
PORT = 10001
HOST = "localhost"
msg = "Hello, World!"
my_socket = UDPSocket.new
my_socket.connect(HOST, PORT)
my_socket.send(msg, 0)
```
Example: UDP Client (2)

```ruby
# File: send_udp.rb
require 'socket'
PORT = 10001
HOST = "localhost"
msg = "Hello, World!"
UDPSocket.new.send(msg, 0, HOST, PORT)
```
UDP Sockets: the Methods

Class Methods:

**UDPSocket.new**(family = AF_INET) -> aSession

**UDPSocket.open**(family = AF_INET) -> aSession

Instance Methods:

**aSession.bind**(hostName, port) -> 0

**aSession.connect**(hostname, port) -> 0

**aSession.recvfrom**(len [, flags]) -> anArray

**aSession.send**(aString, flags) -> aFixnum

**aSession.send**(aString, flags, hostName, port) -> aFixnum
The program flow in the sender and receiver programs

**sender program**

1. create a datagram socket and bind it to any local port;
2. place data in a byte array;
3. create a datagram packet, specifying the data array and the receiver’s address;
4. invoke the send method of the socket with a reference to the datagram packet;

**receiver program**

1. create a datagram socket and bind it to a specific local port;
2. create a byte array for receiving the data;
3. create a datagram packet, specifying the data array;
4. invoke the receive method of the socket with a reference to the datagram packet;
Exercises

Write a Datagram class that would allow recv_udp.rb (slide 10) to be written like this:

```ruby
require 'socket'
require 'datagram'

sock = UDPSocket.new
sock.bind('localhost', inport)
dg = Datagram.new(sock.recvfrom(80))
print "contents: #{dg.contents}\n"
print "sender: #{dg.name} (#{dg.address}) on port #{dg.port}\n"
```
A simple application:

A client reads a line from its standard input and sends the line out its socket to the server.

The server reads a line from its socket.

The server converts the line to uppercase.

The server sends the modified line out its sockets to the client.

The client reads the modified line through its socket and prints the line on its standard output.

Exercises
class Datagram
  def initialize(information)
    @contents = information[0]
    @protocol, @port, @name, @address = information[1]
  end

  attr_reader :contents, :protocol, :port, :name, :address
end
# uc_client.rb
require 'socket'
require 'timeout'
$port = 12345
$host = 'localhost'
client = UDPSocket.new
client.connect($host, $port)
msg = gets.chomp
client.send(msg, 0)

timeout(10) do
  msg_rec = client.recvfrom(80)
  puts msg_rec[0]
  puts msg_rec[1]
end

# uc_server.rb
require 'socket'
$port = 12345
$host = 'localhost'
server = UDPSocket.new
server.bind("localhost", $port)

loop do
  msg = server.recvfrom(125)
  puts "message from #{msg[1][2]} on port #{msg[1][1]}: '#{msg[0]}"
  server.send(msg[0].upcase, 0, msg[1][2], msg[1][1])
end
TCP Sockets

The datagram socket API supports the exchange of discrete units of data (that is, datagrams).

The stream socket API provides a model of data transfer based on the stream-mode I/O of the Unix operating systems.

By definition, a stream-mode socket supports connection-oriented communication only.
Connection-Oriented Socket API

- **P1**
- **P2**

**a data stream**

- **a stream-mode data socket**
- **process**
- **write operation**
- **read operation**

21
A stream-mode socket is established for data exchange between two specific processes.

Data stream is written to the socket at one end, and read from the other end.

A stream socket cannot be used to communicate with more than one process.
In Ruby, the stream-mode socket API is provided with two classes:

- Server socket (**class** TCPServer): for accepting connections; we will call an object of this class a **connection socket**.

- Socket (**class** TCPSocket): for data exchange; we will call an object of this class a **data socket**.
connection listener (server)

create a connection socket
and listen for connection
requests;
accept a connection;
create a data socket for reading from
or writing to the socket stream;
get an input stream for reading
to the socket;
read from the stream;
get an output stream for writing
to the socket;
write to the stream;
close the data socket;
close the connection socket.

connection requester (server)

create a data socket
and request for a connection;
get an output stream for writing
to the socket;
write to the stream;
get an input stream for reading
to the socket;
read from the stream;
close the data socket.
The Server (the connection listener)

A server uses two sockets: one for accepting connections, another for send/receive.
Connection-Oriented Socket API

1. Server establishes a socket sd1 with local address, then listens for incoming connection on sd1.

2. Server accepts the connection request and creates a new socket sd2 as a result.

Client establishes a socket with remote (server's) address.


5. When the protocol has completed, server closes sd2; sd1 is used to accept the next connection.

Client issues send operation.

Client closes its socket when the protocol has completed.
Using TCP Socket

Send a line of text via a TCP socket

```ruby
# File: send_tcp.rb
require "socket"
port = 10000
my_socket = TCPSocket.new("localhost", port)
my_socket.puts "Hello, World."
my_socket.close
```

We could have substituted `File.new(“textfile”, “w”)` for:
# File: recv_tcp.rb

```ruby
require 'socket'

port = 10000

listener = TCPServer.new(port)

my_socket = listener.accept

input = my_socket.gets

my_socket.close

print "Message received: #{input}\n"
```

TCPServer object makes a port available to the outside world.

It listens via the accept method for an incoming connection.

When `accept` returns, somebody has requested a connection, so there is a socket to read and write.
Using TCP Socket

It is important that `recv_tcp.rb` should be the first to start running.

```
$ ruby recv_tcp.rb &
[1] 347
Message received: Hello, World.
$
```

```
$ ruby send_tcp.rb
$
```
A chat application: chat_server.rb

```ruby
require 'socket'
$port = 12345
$host = "localhost"

listener = TCPServer.new($port)
session = listener.accept
listener.close

session.puts "Start talking. It's your turn."

begin
  loop do
    in_msg = session.gets
    print ">>> ", in_msg

    out_msg = STDIN.gets; break if out_msg.nil?
    session.print out_msg
  end
ensure
  session.close
end
```
require 'socket'
$port = 12345
$host = 'localhost'

session = TCPSocket.new($host, $port)

begin
  loop do
    in_msg = session.gets
    print ">>> ", in_msg

    out_msg = STDIN.gets; break if out_msg.nil?
    session.puts out_msg
  end
end

ensure
  session.close
end
Threaded Servers

```ruby
require 'socket'
PORT = 12321
server = TCPServer.new(PORT)
while (ses = server.accept)
  Thread.new(ses) do |my_ses|
    my_ses.puts Time.now
    my_ses.close
  end
end
```