Chapter 1

Sockets

Written by N. Bouraqadi Saâdani and L. Fabresse

Modern software often involve multiple devices that collaborate through a network. The basic approach to set up such collaborations is to use *sockets*. A typical use is in the World Wide Web. Browsers and servers interact through sockets that carry HTTP requests and responses.

The concept of socket was first introduced by researchers from Berkeley University in the 1960s. They defined the first socket API for the C programming language in the context of Unix operating systems. Since then, the concept of socket spread out to other operating systems. Its API was ported to almost all programming languages.

In this chapter, we present the API of sockets in the context of Pharo. We first show through some examples how to use sockets for building both clients and servers. The notion of client and server are inherent to sockets: a server waits for requests emitted by clients. Then, we introduce SocketStream and how to use it. In practice, one is likely to use SocketStream instead of plain sockets. The chapter ends with a description of some unix networking utilities that are useful for experimenting.

1.1 Basic Concepts

Socket

A remote communication involves at least two system processes exchanging some data bytes through a network. Each process accesses the network through at least one socket (see Figure 1.1). A socket can then be defined

as a *plug on a communication network*.

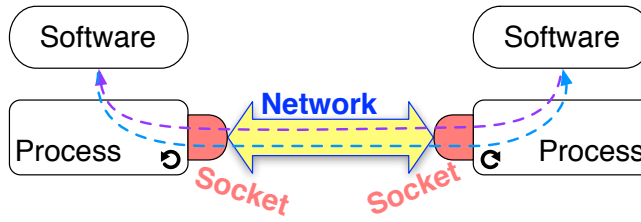

Figure 1.1: Inter-Process Remote Communication Through Sockets

Sockets are used to establish a bidirectional communication: they allow both sending and receiving data. Such interaction can be done according to communication protocols which are encapsulated by sockets. On the Internet and other networks such as ethernet LANs[1], two basic protocols widely used are *TCP/IP* and *UDP/IP*.

TCP/IP vs. UDP/IP

TCP/IP stands for *Transmission Control Protocol / Internet Protocol* (TCP for short). TCP use guarantees a reliable communication (no data loss). It requires that applications involved in the communication get connected before actually communicating. Once a connection is established interacting parties can send and receive an arbitrary amount of bytes. This is often referred to as a *stream communication*. Data reach the destination in the same order of their sending.

UDP/IP stands for *User Datagram Protocol / Internet Protocol* (UDP for short). Datagrams are chunks of data which size cannot exceed 64KB. UDP is an unreliable protocol because of two reasons. First, UDP does not guarantee that datagrams will actually reach their destination. The second reason is that the reception order of multiple datagrams from a single sender to the receiver may arrive in an arbitrary order. Nevertheless, UDP is faster than TCP since no connection is required before sending data. A typical use of UDP is "heart-beating" as used in server-based social application, where clients need to notify the server their status (*e.g.*, Requesting interactions, or Invisible).

In the remainder of this chapter we will focus on TCP Sockets. First, we show how to create a client socket, connect it to a server, exchange data and close the connection (Section 1.2). This lifecycle is illustrated using examples showing the use of client sockets to interact with a web server.

[1] Local Area Networks.

Next, Section 1.3 presents server sockets. We describe their life-cycle and how to use them to implement a server that can handle concurrent connections. Last, we introduce in Section 1.4 socket streams. We give an overview of their benefits by describing their use on both client and server side.

1.2 TCP Client

We call *TCP client* an application that initiates a TCP connection to exchange data with another application: the *server*. It is important to mention that the client and the server may be developed in different languages. The life-cycle of such a client in Pharo decomposes into 4 steps:

1. Create a TCP socket.
2. Connect the socket to a server.
3. Exchange data with the server through the socket.
4. Close the socket.

Create a TCP Socket

Pharo provides a single socket class. It has one creation method per socket type (TCP or UDP). To create a TCP socket, you need to evaluate the following expression:

```
Socket newTCP
```

Connect a TCP Socket to some Server

To connect a TCP Socket to a server, you need to have the object representing the IP address of that server. This address is an instance of SocketAddress. A handy way to create it is to use NetNameResolver that provides IP style network name lookup and translation facilities.

Script 1.1 provides two examples of socket address creation. The first one creates an address from a string describing the server name ('www.esug.org'), while the second does the creation from a string representing the IP address of the server ('127.0.0.1'). Note that to use the NetNameResolver you need to have your machine connected to a network with a DNS[2],

[2] *Domain Name System*: basically a directory that maps device names to their IP address.

which should probably be the case. The only exception is for retrieving the local host address, i.e. 127.0.0.1 which is the generic address to refer to the machine that runs your software (Pharo here).

Script 1.1: *Creating a Socket Address*

```
| esugAddress localAddress |
esugAddress := NetNameResolver addressForName: 'www.esug.org'.
localAddress := NetNameResolver addressForName: '127.0.0.1'.
```

Now we can connect our TCP socket to the server as shown in Script 1.2. Message connectTo:port: attempts to connect the socket to the server using the server address and port provided as parameters. The server address refers to the address of the network interface (e.g. ethernet, wifi) used by the server. The port refers to the communication endpoint on the network interface. Each network interface has for each IP transport protocol (e.g. TCP, UDP) a collection of ports that are numbered from 0 to 65535. For a given protocol, a port number on an interface can only be used by a single process.

Script 1.2: *Connecting a TCP Socket to ESUG Server.*

```
| clientSocket serverAddress |
clientSocket := Socket newTCP.
serverAddress := NetNameResolver addressForName: 'www.esug.org'.
clientSocket
   connectTo: serverAddress port: 80;
   waitForConnectionFor: 10.
clientSocket isConnected
   ⟶    true
```

The connectTo:port: message returns immediately after issuing to the system (through a primitive call) the request to connect the socket. Message waitForConnectionFor: 10 suspends the current process until the socket is connected to the server. It waits at most 10 seconds as requested by the parameter. If the socket is not connected after 10 seconds, the ConnectionTimedOut exception is signaled. Otherwise, the execution can proceed by evaluating the expression clientSocket isConnected which obviously answers true.

Exchange Data with Server

Once the connection is established, the client can exchange (send/receive) instances of ByteString with the server. Typically, the client sends some request to the server and then expects for a response. Web browsers act according to this schema. A web browser is a client that issues a request to some web server identified by the URL. Such request is often the path

to some resource on the server such as a html file or a picture. Then, the browser awaits the server response (*e.g.*, html code, picture bytes).

Script 1.3: *Exchanging Data with some Server through a TCP Socket.*

```
| clientSocket data |
... "create and connect the TCP clientSocket"
clientSocket sendData: 'Hello server'.
data := clientSocket receiveData.
... "Process data"
```

Script 1.3 shows the protocol to send and receive data through a client socket. Here, we send the string 'Hello server!' to the server using the sendData: message. Next, we send the receiveData message to our client socket to read the answer. Note that reading the answer is *blocking*, meaning receiveData returns when a response has been read. Then, the contents of variable data is processed.

Script 1.4: *Bounding the Maximum Time for Data Reception.*

```
|clientSocket data|
... "create and connect the TCP clientSocket"
[data := clientSocket receiveDataTimeout: 5.
... "Process data"
  ] on: ConnectionTimedOut
  do: [ :timeOutException |
      self
          crLog: 'No data received!';
          crLog: 'Network connection is too slow or server is down.']
```

Note that by using receiveData, the client waits until the server either sends no more data, or closes the connection. This means that the client may wait indefinitely. An alternative is to have the client signal a ConnectionTimedOut exception if it had waited too much as shown in Script 1.4. We use message receiveDataTimeout: to ask the client socket to wait for 5 seconds. If data is received during this period of time, it is processed silently. But if no data is received during the 5 seconds, a ConnectionTimedOut is signaled. In the example we log a description of what happened.

Close a Socket

A TCP socket remains alive while devices at both ends are connected. A socket is closed by sending the message close to it. The socket remains connected until the other side closes it. This may last indefinitely when there is a network failure or when the other side is down. This is why sockets

also accept the destroy message, which frees system resources required by the socket.

In practice we use closeAndDestroy. It first attempts to close the socket by sending the close message. Then, if the socket is still connected after a duration of 20 seconds, the socket is destroyed. Note that there exist a variant closeAndDestroy: seconds which takes as a parameter the duration to wait before destroying the socket.

Script 1.5: *Interaction with a Web Site and Cleanup.*

```
| clientSocket serverAddress httpQuery htmlText |
httpQuery := 'GET / HTTP/1.1', String crlf,
   'Host: www.esug.org:80', String crlf,
   'Accept: text/html', String crlfcrlf.
serverAddress := NetNameResolver addressForName: 'www.esug.org'.
clientSocket := Socket newTCP.
[ clientSocket
   connectTo: serverAddress port: 80;
   waitForConnectionFor: 10.
 clientSocket sendData: httpQuery.
 htmlText := clientSocket receiveDataTimeout: 5.
 htmlText crLog ] ensure: [clientSocket closeAndDestroy].
```

To summarize all steps described so far, we use the example of getting a web page from a server in Script 1.5. First, we forge a HTTP[3] query. The string corresponding to our query starts with the GET keyword, followed by a slash saying that we are requesting the root file of the server. Follows the protocol version HTTP/1.1. The second line includes the name of the web server and its port. The third and last line of the HTTP query refers to format accepted by our client. Since, we intend to display the result of our query on the Transcript, we state in the HTTP query (see line beginning with Accept:) that our client accepts texts with html format.

Next, we retrieve the IP address of the www.esug.org server. Then, we create a TCP socket and connect it to the server. We use the IP address we get in the previous step and the default port for web servers: 80. The connection should be established in less than 10 seconds (waitForConnectionFor: 10), otherwise we get a ConnectionTimedOut exception.

After sending the http query (clientSocket sendData: httpQuery), we read form the socket the received html text that we display. Note that the we ask the socket to wait at most 5 seconds for the answer of the server (clientSocket receiveDataTimeout: 5). On timeout, the socket answers an empty socket.

Finally, we close the socket and free related resources (clientSocket closeAndDestroy). We ensure the clean up by means of the ensure: message

[3]HyperText Transfer Protocol used for web communications.

sent to the block that performs socket connection and data exchange with the web server.

1.3 TCP Server

Now, let us build a simple TCP server. A *TCP Server* is an application that awaits TCP connections from TCP clients. Once connection established, both the server and the client can send a receive data in any order. A big difference between the server and the client is that the server uses at least two sockets. One socket is used for handling client connections, while the second serves for exchanging data with a particular client.

TCP Socket Server Life-cycle

The life-cycle of a TCP server has 5 steps:

1. Create a first TCP socket labelled *connectionSocket*.
2. Wait for a connection by making *connectionSocket* listen on a port.
3. Accept a client request for connection. As a result, *connectionSocket* will build a second socket labelled *interactionSocket*.
4. Exchange data with the client through *interactionSocket*. In the meanwhile, *connectionSocket* can continue to wait for a new connection, and possibly create new sockets to exchange data with other clients.
5. Close *interactionSocket*.
6. Close *connectionSocket* when we decide to kill the server and stop accepting client connections.

Concurrency of this life-cycle is made explicit on Figure 1.2. The server listens for incoming client connection requests through *connectionSocket*, while exchanging data with possibly multiple clients through multiple *interactionSockets* (one per client). In the following, we first illustrate the socket serving machinery. Then, we describe a complete server class and explain the server life-cycle and related concurrency issues.

Serving Basic Example

We illustrate the serving basics through a simple example of an echo TCP server that accepts a single client request. It sends back to clients whatever data it received and quits. The code is provided by Script 1.6.

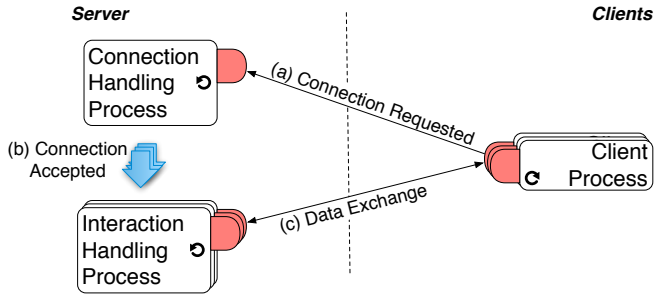

Figure 1.2: Socket Server Concurrently Servers Multiple Clients

Script 1.6: *Basic Echo Server.*

```
| connectionSocket interactionSocket receivedData |
"Prepare socket for handling client connection requests"
connectionSocket := Socket newTCP.
connectionSocket listenOn: 9999 backlogSize: 10.

"Build a new socket for interaction with a client which connection request is accepted"
interactionSocket := connectionSocket waitForAcceptFor: 60.

"Get rid of the connection socket since it is useless for the rest of this example"
connectionSocket closeAndDestroy.

"Get and display data from the client"
receivedData := interactionSocket receiveData.
receivedData crLog.

"Send echo back to client and finish interaction"
interactionSocket sendData: 'ECHO: ', receivedData.
interactionSocket closeAndDestroy.
```

First, we create the socket that we will use for handling incoming connections. We configure it to listen on port 9999. The backlogSize is set to 10, meaning that we ask the Operating System to allocate a buffer for 10 connection requests. This backlog will not be actually used in our example. But, a more realistic server will have to handle multiple connections and then store pending connection requests into the backlog.

Once the connection socket (referenced by variable connectionSocket) is set up, it starts listening for client connections. The waitForAcceptFor: 60 message makes the socket wait connection requests for 60 seconds. If no client attempts to connect during these 60 seconds, the message answers nil. Otherwise, we get a new socket interactionSocket connected the client's socket.

At this point, we do not need the connection socket anymore, so we can close it (connectionSocket closeAndDestroy message).

Since the interaction socket is already connected to the client, we can use it to exchange data. Messages receiveData and sendData: presented above (see Section 1.2) can be used to achieve this goal. In our example, we wait for data from the client, next we display it on the Transcript, and last we send it back to the client prefixed with the 'ECHO: ' string. Last, we finish the interaction with the client by closing the interaction socket.

There are different options to test the server of Script 1.6. The first simple one is to use the nc (netcat) utility discussed in Section 1.5. First run the server script in a workspace. Then, in a terminal, evaluate the following command line:

```
echo "Hello Pharo" | nc localhost 9999
```

As a result, on the Transcript of the Pharo image, the following line should be displayed:

```
Hello Pharo
```

On the client side, that is the terminal, you should see:

```
ECHO: Hello Pharo
```

A pure Pharo alternative relies on using two different images: one that runs the server code and the other for client code. Indeed, since our examples run within the user interaction process, the Pharo UI will be frozen at some points, such as during the waitForAcceptFor:. Script 1.7 provides the code to run on the client image. Note that you have to run the server code first. Otherwise, the client will fail. Note also that after the interaction, both the client and the server terminate. So, if you want to run the example a second time you need to run again both sides.

Script 1.7: *Echo Client.*

```
| clientSocket serverAddress echoString |
serverAddress := NetNameResolver addressForName:'127.0.0.1'.
clientSocket := Socket newTCP.
[ clientSocket
    connectTo: serverAddress port: 9999;
    waitForConnectionFor: 10.
 clientSocket sendData: 'Hello Pharo!'.
 echoString  := clientSocket receiveDataTimeout: 5.
 echoString crLog.
] ensure: [ clientSocket closeAndDestroy ].
```

Echo Server Class

We define here the EchoServer class that deals with concurrency issues. It handles concurrent client queries and it does not freeze the UI. Figure 1.3 shows an example of how the EchoServer handles two clients.

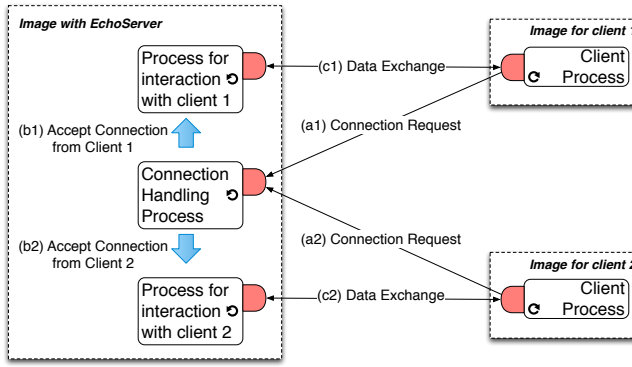

Figure 1.3: Echo Server Concurrently Serving Two Clients

As we can see in the definition labelled class 1.8, the EchoServer declares three instance variables. The first one (connectionSocket) refers to the socket used for listening to client connections. The two last instance variables (isRunning holding a boolean and isRunningLock holding a Mutex) are used to manage the server process life-cycle while dealing with synchronization issues.

Class 1.8: EchoServer *Class Definition*

```
Object subclass: #EchoServer
   instanceVariableNames: 'connectionSocket isRunning isRunningLock'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'SimpleSocketServer'
```

The isRunning instance variable is a flag that is set to true while the serving is running. As we will see below, it can be accessed by different processes. Therefore, we need to ensure that read accesses to the flag get a coherent value even if there are concurrent write accesses. This is achieved using a lock (isRunningLock instance variable) that guarantees that isRunning is accessed by only by a single process each time.

Method 1.9: *The* EchoServer»isRunning *Read Accessor*

```
EchoServer»isRunning
   ^ isRunningLock critical: [ isRunning ]
```

Method 1.10: *The* EchoServer»isRunning: *Write Accessor*

```
EchoServer»isRunning: aBoolean
  isRunningLock critical: [ isRunning := aBoolean ]
```

Accesses to the flag are only possible through accessor methods (method 1.9 and method 1.10). Thus, isRunning is read and wrote inside blocks that are arguments of message critical: sent to isRunningLock. This lock is an instance of Mutex (see method 1.11). When receiving a critical: message, a mutex evaluates the argument (a block). During this evaluation, other processes that send a critical: message to the same mutex are suspended. Once the first block is done, the mutex resumes a suspended process (the one that was first suspended). This cycle is repeated until there are no more suspended processes. Thus, the mutex ensures that the isRunning flag is read and wrote sequentially.

Method 1.11: *The* EchoServer»initialize *Method*

```
EchoServer»initialize
  super initialize.
  isRunningLock := Mutex new.
  self isRunning: false
```

To manage the life-cycle of our server, we introduced two methods EchoServer»start and EchoServer»stop. We begin with the simplest one EchoServer»stop which definition is provided as method 1.12. It simply sets the isRunning flag to false. This will have the consequence of stopping the serving loop in method EchoServer»serve (see method 1.13).

Method 1.12: *The* EchoServer»stop *Method*

```
EchoServer»stop
  self isRunning: false
```

Method 1.13: *The* EchoServer»serve *Method*

```
EchoServer»serve
  [ [ self isRunning ]
    whileTrue: [ self interactOnConnection ] ]
    ensure: [ connectionSocket closeAndDestroy ]
```

The activity of the serving process is implemented in the serve method (see method 1.13). It interacts with clients on connections while the isRunning flag is true. After a stop, the serving process terminates by destroying the connection socket. The ensure: message guarantees that this destruction is performed even if the serving process is terminated abnormally. Such termination may occur because of an exception (*e.g.*, network disconnection) or a user action (*e.g.*, through the process browser).

Method 1.14: *The* EchoServer»start *Method*

```
EchoServer»start
   isRunningLock critical: [
      self isRunning ifTrue: [ ^ self ].
      self isRunning: true].
   connectionSocket := Socket newTCP.
   connectionSocket listenOn: 9999 backlogSize: 10.
   [ self serve ] fork
```

The creation of the serving process is the responsibility of method EchoServer»start (see the last line of method 1.14). The EchoServer»start method first checks wether the server is already running. It returns if the isRunning flag is set to true. Otherwise, a TCP socket dedicated to connection handling is created and made to listen on port 9999. The backlog size is set to 10 that is -as mentioned above- the system allocates a buffer for storing 10 pending client connection requests. This value is a trade-off that depends on how fast the server is (depending on the VM and the hardware) and the maximum rate of client connections requests. The backlog size has to be large enough to avoid losing any connection request, but not too big to avoid wasting memory. Finally EchoServer»start method creates a process by sending the fork message to the [self serve] block. The created process has the same priority as the creator process (i.e., the one that performs the EchoServer»start method, the UI process if you have executed it from a workspace).

Method 1.15: *The* EchoServer»interactOnConnection *Method*

```
EchoServer»interactOnConnection
   | interactionSocket |
   interactionSocket := connectionSocket waitForAcceptFor: 1 ifTimedOut: [^self].
   [self interactUsing: interactionSocket] fork
```

Method EchoServer»serve (see method 1.13) triggers interactions with connected clients. This interaction is handled in the EchoServer» interactOnConnection method (see method 1.15). First, the connection socket waits for client connections for one second. If no client attempts to connect during this period we simply return. Otherwise, we get as result another socket dedicated to interaction. To process other client connection requests, the interaction is performed in another process, hence the fork in the last line.

Method 1.16: *The* EchoServer»interactUsing: *Method*

```
EchoServer»interactUsing: interactionSocket
   | receivedData |
   [ receivedData := interactionSocket receiveDataTimeout: 5.
```

```
    receivedData crLog.
    interactionSocket sendData: 'ECHO: ', receivedData
  ] ensure: [
    interactionSocket closeAndDestroy ]
```

The interaction as implemented in method EchoServer»interactUsing: (see method 1.16) with a client boils down to reading data provided by the client and sending it back prefixed with the 'ECHO: ' string. It is worth noting that we ensure that the interaction socket is destroyed, whether we have exchanged data or not (timeout).

1.4 SocketStream

SocketStream is a read-write stream that encapsulates a TCP socket. It eases the data exchange by providing buffering together with a set of facility methods. It provides an easy-to-use API on top of Socket.

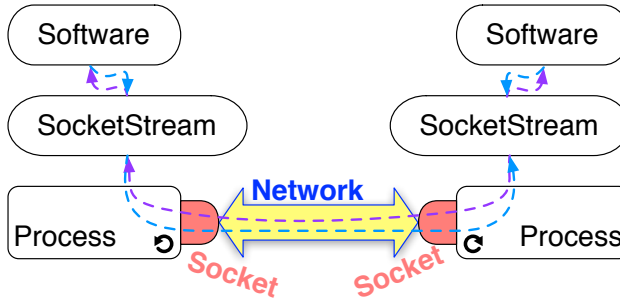

Figure 1.4: SocketStream allows to use Socket in an easy way.

A SocketStream can be created using the method SocketStream class>>openConnectionToHost:port:. By providing the host address or name and the port, it initialises a new Socket with the default parameter of the system. But, you can build a SocketStream on top of an existing Socket with the method SocketStream class>>on:, which allows you to send data on a socket you have already configured.

With your new SocketStream, you can receive data with the useful methods receiveData, which waits data until a timeout, or receiveAvailableData, which receives data but does not wait for more to arrive. The method isDataAvailable allows you to check if data is available on the stream before receiving it.

You can send data using the methods nextPut:, nextPutAll:, or nextPutAllFlush: that put data in the stream. The method nextPutAllFlush: flushes the other pending data before putting the data in the stream.

Finally, when the SocketStream is finished to use, send the message close to finish and close the associated socket. Other useful methods of SocketStream are explained below in the chapter.

SocketStream at Client Side

We illustrate socket stream use at client side with the following code snippet (Script 1.17). It shows how the client uses a socket stream to get the first line of a webpage.

Script 1.17: *Getting the first line of a web page using* SocketStream.

```
| stream httpQuery result |
stream := SocketStream
      openConnectionToHostNamed: 'www.pharo-project.org'
      port: 80.
httpQuery := 'GET / HTTP/1.1', String crlf,
   'Host: www.pharo-project.org:80', String crlf,
   'Accept: text/html', String crlf.
[ stream sendCommand: httpQuery.
stream nextLine crLog ] ensure: [ stream close ]
```

The first line creates a stream that encapsulates a newly created socket connected to the provided server. It is the responsibility of message openConnectionToHostNamed:port:. It suspends the execution until the connection with the server is established. If the server does not respond, the socket stream signals a ConnectionTimedOut exception. This exception is actually signaled by the underlying socket. The default timeout delay is 45 seconds (defined in method Socket class»standardTimeout). One can choose a different value using the SocketStream»timeout: method.

Once our socket stream is connected to the server, we forge and send an HTTP GET query. Notice that compared to script 1.5 (page 6) we skipped here (Script 1.17) one final String crlf. This is because the SocketStream»sendCommand: method automatically inserts CR and LF characters after sent data to mark line ending.

Reception of the requested web page is triggered by sending the nextLine message to our socket stream. It will wait for a few seconds until data is received. Data is then displayed on the transcript. We safely ensure that the connection is closed.

In this example, we only display the first line of response sent by the server. We can easily display the full response including the html code by sending the upToEnd message to our socket stream. Note however that you will have to wait a bit longer compared to displaying a single line.

SocketStream at Server Side

SocketStreams may also be used at server side to wrap the interaction socket as shown in Script 1.18.

Script 1.18: *Simple Server using* SocketStream.

```
| connectionSocket interactionSocket interactionStream |
connectionSocket := Socket newTCP.
[
   connectionSocket listenOn: 12345 backlogSize: 10.
   interactionSocket := connectionSocket waitForAcceptFor: 30.
   interactionStream := SocketStream on: interactionSocket.
   interactionStream sendCommand: 'Greetings from Pharo Server'.
   interactionStream nextLine crLog.
] ensure: [
   connectionSocket closeAndDestroy.
   interactionStream ifNotNil: [interactionStream close]
]
```

A server relying on socket streams still uses a socket for handling incoming connection requests. Socket streams come into action once a socket is created for interaction with a client. The socket is wrapped into a socket stream that eases data exchange using messages such as sendCommand: or nextLine. Once we are done, we close and destroy the socket handling connections and we close the interaction socket stream. This latter will take care of closing and destroying the underlying interaction socket.

Binary vs. Ascii mode

Data exchanged can be treated as bytes or characters. When a socket stream is configured to exchange bytes using binary, it sends and receives data as byte arrays. Conversely, when a socket stream is configured to exchange characters (default setting) using message ascii, it sends and receives data as Strings.

Suppose we have an instance of the EchoServer (see Section 1.3) started by means of the following expression

```
server := EchoServer new.
server start.
```

The default behavior of socket stream is to handle ascii strings on sends and receptions. We show instead in Script 1.19 the behavior in binary mode. The nextPutAllFlush: message receives a byte array as argument. It puts all the bytes into the buffer then immediately triggers the sending (hence the Flush in the message name). The upToEnd message answers an array with all bytes sent back by the server. Note that this message blocks until the connection with the server is closed.

Script 1.19: *A* SocketStream *Interacting in Binary Mode.*

```
interactionStream := SocketStream
                        openConnectionToHostNamed: 'localhost'
                        port: 9999.
interactionStream binary.
interactionStream nextPutAllFlush: #[65 66 67].
interactionStream upToEnd.
```

Note that whether the client manages strings (ascii mode) or byte arrays (binary mode) has no impact on the server. Indeed, in ascii mode, the socket stream handles instances of ByteString. So, each character maps to a single byte.

Delimiting Data

SocketStream acts simply as a gateway to some network. It sends or reads bytes without giving them any semantics. The semantics, that is the organization and meaning of exchanged data should be handled by other objects. Developers should decide a protocol to use and to enforce on both interacting sides to have correct interaction.

A good practice is to *reify* a protocol, that is to materialize it as an object which wraps a socket stream. The protocol object analyzes exchanged data and decides accordingly which messages to send to the socket stream. Involved entities in any conversation need a protocol that defines how to organize data into sequence of bytes or characters. Senders should conform to this organization to allow receivers to extract valid data from received sequence of bytes.

One possible solution is to have a set of delimiters inserted between bytes or characters corresponding to each data. An example of delimiter is the sequence of ASCII characters CR and LF. This sequence is considered so useful that the developers of the SocketStream class introduced the sendCommand: message. This method (illustrated in script 1.5) appends CR and LF after sent data. When reading CR followed by LF the receiver knows that the received sequence of characters is complete and can be safely converted into valid data. A facility method nextLine (illustrated in

script 1.17) is implemented by SocketStream to perform reading until the reception of CR+LF sequence. One can however use any character or byte as a delimiter. Indeed, we can ask a socket stream to read all characters/bytes up to some specific one using the upTo: message.

The advantage of using delimiters is that it handles data of arbitrary size. The cons is that we need to analyze received bytes or characters to find out the limits, which is resource consuming. An alternative approach is to exchange bytes or characters organized in chunks of a fixed size. A typical use of this approach is for streaming audio or video contents.

Script 1.20: *A content streaming source sending data in chunks.*

```
interactionStream := "create an instance of SocketStream".
contentFile := FileStream fileNamed: '/Users/noury/Music/mySong.mp3'.
contentFile binary.
content := contentFile upToEnd.
chunkSize := 3000.
chunkStartIndex := 1.
[chunkStartIndex < content size] whileTrue: [
   interactionStream next: chunkSize putAll: content startingAt: chunkStartIndex.
   chunkStartIndex := chunkStartIndex + chunkSize.
]
interactionStream flush.
```

Script 1.20 gives an example of a script streaming an mp3 file. First we open a binary (mp3) file and retrieve all its content using the message upToEnd:. Then we loop, sending data in chunks of 3000 bytes. We rely on the next:putAll:startingAt: message that takes three arguments: the size (number of bytes or characters) of the data chunk, the data source (a sequenceable collection) and the index of the first element of the chunk. In this example, we make the assumption that the size of the content collection is a multiple of the chunk size. Of course, in a real setting, this assumption does not hold and one needs to deal with the last part of data that is smaller than a chunk. A possible solution is to replace missing bytes with zeros. In addition, loading everything in memory first is often not a practical solution and streaming approaches are usually used instead.

Script 1.21: *Reading data in chunks using* SocketStream.

```
| interactionStream chunkSize chunk |
interactionStream := SocketStream
                        openConnectionToHostNamed: 'localhost'
                        port: 9999.
interactionStream isDataAvailable ifFalse: [(Delay forMilliseconds: 100) wait].
chunkSize := 5.
[interactionStream isDataAvailable] whileTrue: [
   chunk  :=  interactionStream next: chunkSize.
```

```
    chunk crLog.
].
interactionStream close.
'DONE' crLog.
```

To read data in chunks, SocketStream responds to the next: message as illustrated by script 1.21. We consider that we have a server running at port 9999 of our machine that sends a string which size is a multiple of 5. Right after the connection, we wait 100 milliseconds until the data is received. Then, we read data in chunks of 5 characters that we display on the Transcript. So, if the server sends a string with 10 characters 'HelloWorld', we will get on the Transcript Hello on one line and World on a second line.

1.5 Tips for Networking Experiments

In sections related to client-side sockets and socket streams, we used interactions with a web server as an example. So, we forged an HTTP Get query and send it to the server. We chose these examples to make experiments straightforward and platform agnostic. In real scale applications, interactions involving HTTP should be coded using a higher level library such as Zinc HTTP Client/Server library that is part of the default Pharo distribution[4].

Network programming can easily scale up in complexity. Using a toolbox outside Pharo is often necessary to identify what is the source of an odd behavior. This section lists a number of Unix utilities to deal with low level network operations. Readers with a Unix machine (Linux, Mac OS X) or with Cygwin (for Windows) can use nc (or netcat), netstat and lsof for their tests.

nc (netcat)

nc allows one to set up either a client or a server for both TCP (default protocol) and UDP. It redirects the content of its stdin to the other side. The following snippet show how to send 'Hello from a client' to a server on the local machine listening on port 9090.

```
echo Hello from a client | nc 127.0.0.1 9090
```

The command line below starts a server listening on port 9090 that sends 'Hi from server' to the first client to connect. It terminates after the interaction.

[4] http://zn.stfx.eu/zn/index.html

```
echo Hi from server | nc -l 9090
```

You can keep the server running by means of option `-k`. But, the string produced by the preceding `echo` is sent only to the first client to connect. An alternative solution is to make the `nc` server send text while you type. Simply evaluate the following command line:

```
echo nc -lk 9090
```

Type in some text in the same terminal where you started the server. Then, run a client in another terminal. Your text will be displayed on the client side. You can repeat these two last actions (type text at the server side, then start client) as many times as needed.

You can even go more interactive by making the connection between a client and a server more persistent. By evaluating the following command line, the client sends every line (ended with "Enter"). It will terminate when sending the EOF signal (ctl-D).

```
echo cat | nc -l 9090
```

netstat

This command provides various information on network interfaces and sockets of your computer. It provides many statistics so one needs to use appropriate options to filter out useful information. The following command line allows displaying status of tcp sockets and their addresses. Note that the port numbers and addresses are separated by a dot.

```
netstat -p tcp -a -n
```

lsof

The `lsof` command lists all files open in your system. This of course includes sockets, since everything is a file in Unix. Why `lsof` is useful, would you say, if we already have `netstat`? The answer is that `lsof` shows the link between processes and sockets. So, you can find out sockets related to your program.

The example provided by following command line lists TCP sockets. The `n` and `P` options force `lsof` to display host addresses and ports as numbers.

```
lsof -nP -i tcp
```

1.6 Chapter summary

This chapter introduces the use TCP sockets and socket streams to develop both network clients and servers. It has reviewed the survival kit of network programming:

- Sockets are low-level bi-directional communication gateways instances of class Socket.
- Socket-based programming always involves one server and one or more clients.
- A server waits for requests emitted by clients.
- Messages sendData: and receiveData are the socket primitives to send and receive data.
- A maximum waiting time can be set using receiveDataTimeout:.
- SocketStream is a buffered read-write stream that encapsulates a TCP socket.
- Network DNS is accessible through the class NetNameResolver, which converts device names into numerical internet addresses.
- Unix does provide some networking utilities that are useful for debugging and testing purposes.

As mentioned in the introduction, we recommend to use socket streams which are of higher level and provide facility methods. They were successfully used in projects such as Swazoo and Kom web servers used respectively by AidaWeb and Seaside web frameworks.

Nevertheless, socket streams remain still low-level if you have an application involving different objects distributed over communicating images. In a such software, developers need to deal with message passing between remote objects by serializing arguments and results. They will have also to take care of distributed garbage-collection. An object should not be destroyed if it is referenced by a remote one. These recurrent not trivial issues are solved by Object-Request Brokers (ORB) such as rST[5]. An ORB frees the developer from networking issues and thus allows expressing remote communications simply using messages exchanged between remote objects.

[5]http://www.squeaksource.com/rST.html