**Лабораторная работа 1**

**Java networking**

Network Programming involves writing programs that communicate with other programs across a computer network.

There are many issues that arise when doing network programming which do not appear when doing single program applications. However, JAVA makes networking applications simple due to the easy-to-use libraries. In general, applications that have components running on different machines are known as distributed applications ... and usually they consist of client/server relationships.

A server is an application that provides a "service" to various clients who request the service.

There are many client/server scenarios in real life:

* Bank tellers (server) provide a service for the account owners (client)
* Waitresses (server) provide a service for customers (client)
* Travel agents (server) provide a service for people wishing to go on vacation (client)

In some cases, servers themselves may become clients at various times.

* E.g., travel agents will become clients when they phone the airline to make a reservation or contact a hotel to book a room.

In the general networking scenario, everybody can either be a client or a server at any time.

This is known as peer-to-peer computing. In terms of writing java applications, it is similar to

having many applications communicating among one another.

* E.g., the original Napster worked this way. Thousands of people all acted as clients (trying to download songs from another person) as well as servers (in that they allowed others to download their songs).

There are many different strategies for allowing communication between applications. JAVA technology allows:

* internet clients to connect to servlets or back-end business systems (or databases).
* applications to connect to one another using sockets.
* applications to connect to one another using RMI (remote method invocation).
* some others

We will look at the simplest strategy of connecting applications using sockets.

A Protocol is a standard pattern of exchanging information.

Computers running on the internet typically use one of the following high-level Application Layer protocols to allow applications to communicate:

* Hyper Text Transfer Protocol (HTTP)
* File Transfer Protocol (FTP)
* Telnet

This is analogous to having multiple strategies for communicating with someone (in person, by phone, through electronic means, by post office mail etc...).

In a lower Transport Layer of communication, there is a separate protocol which is used to determine how the data is to be transported from one machine to another:

* Transport Control Protocol (TCP)
* User Datagram Protocol (UDP)

This is analogous to having multiple ways of actually delivering a package to someone (Email,

Fax, UPS, Fed-Ex etc...)

Beneath that layer is a Network Layer for determining how to locate destinations for the data (i.e., address). And at the lowest level (for computers) there is a Link Layer which actually handles the transferring of bits/bytes. So, internet communication is built of several layers:



When you write JAVA applications that communicate over a network, you are programming in

the Application Layer. JAVA allows two types of communication via two main types of Transport Layer protocols:





One more important definition we need to understand is that of a port:

A port is used as a gateway or "entry point" into an application.

Although a computer usually has a single physical connection to the network, data sent by different applications or delivered to them do so through the use of ports configured on the same physical network connection. When data is to be transmitted over the internet to an application, it requires that we specify the address of the destination computer as well as the application's port number. A computer's address is a 32-bit IP address. The port number is a 16-bit number ranging from 0 to 65,535, with ports 0-1023 restricted by well-known applications like HTTP and FTP.

So, a URL can be used to represent the "location" of a webpage or web-based application. A URL is really just a String that represents the name of a resource ... which can be files, databases, applications, etc. A resource name consists of a host machine name, filename, port number, and other information. It may also specify a protocol identifier (e.g., http, ftp)

Here are some examples of URLs:

* http://www.cnn.com/
* http://www.apple.com/ipad/index.html
* http://en.wikipedia.org/wiki/Computer\_science

Here, http:// is the protocol identifier which indicates the protocol that will be used to obtain the resource. The remaining part is the resource name, and its format depends on the protocol used to access it.

A URL resource name may generally contain:

* a Host Name - The name of the machine on which the resource lives.

<http://www.apple.com:80/ipad/index.html>

* a Port # (optional) - The port number to which to connect.

<http://www.apple.com:80/ipad/index.html>

* a Filename - The pathname to the file on the machine.

<http://www.apple.com:80/ipad/index.html>

In JAVA, there is a URL class defined in the java.net package. We can create our own URL

objects as follows:

URL webPage = new URL("http://www.apple.com/ipad/index.html");

JAVA will "dissect" the given String in order to obtain information about protocol, hostName,

file etc.... Due to this, JAVA may throw a MalformedURLException ... so we will need to do this:

try {

URL webPage = new URL("http://www.apple.com/ipad/index.html");

} catch (MalformedURLException e) {

...

}

Another way to create a URL is to break it into its various components:

try {

URL webPage = new URL("http","www.apple.com",80,"/ipad/index.html");

} catch(MalformedURLException e) {

...

}

If you take a look at the JAVA API, you will notice some other constructors as well. The URL class also supplies methods for extracting the parts (protocol, host, file, port and reference) of a URL object. Here is an example that demonstrates what can be accessed. Note that this example only manipulates a URL object, it does not go off to grab any web pages:

**Exercise 1**

|  |
| --- |
| **import** java.net.\*; **public class** Sockets {  **public static void** main(String[] args) {  URL webpage = **null**;  **try** {  webpage = **new** URL(**"http"**, **"www.apple.com"**, 80, **"/ipad/index.html"**);  } **catch**(MalformedURLException e) {  e.printStackTrace();  }  System.***out***.println(webpage);  System.***out***.println(**"protocol = "** + webpage.getProtocol());  System.***out***.println(**"host = "** + webpage.getHost());  System.***out***.println(**"filename = "** + webpage.getFile());  System.***out***.println(**"port = "** + webpage.getPort());  System.***out***.println(**"ref = "** + webpage.getRef());  } } |

**Exercise 2**

Here is an example that reads a URL directly. It actually reads the file on wikipedia and displays it line by line to the console. Notice that it reads the file as a text file, so we simply get the HTML code. Also, you must be connected to the internet to run this code:

|  |
| --- |
| **import** java.net.\*; **import** java.io.\*; **public class** Sockets {  **public static void** main(String[] args) {  URL wiki = **null**;  **try** {  wiki = **new** URL(**"https://en.wikipedia.org/wiki/Computer\_science"**);  BufferedReader in = **new** BufferedReader(  **new** InputStreamReader(wiki.openStream())); *// Now read the webpage file* String lineOfWebPage;  **while** ((lineOfWebPage = in.readLine()) != **null**)  System.***out***.println(lineOfWebPage);  in.close(); *// Close the connection to the net* } **catch**(MalformedURLException e) {  System.***out***.println(**"Cannot find webpage "** + wiki);  } **catch**(IOException e) {  System.***out***.println(**"Cannot read from webpage "** + wiki);  }  } } |

**Exercise 3**

Here is a modification to the above example that reads the URL by making a URLConnection first. Since the tasks of opening a connection to a webpage and reading the contents may both generate an IOException, we cannot distinguish the kind of error that occurred. By trying to establish the connection first, if any IOExceptions occur, we know they are due to a connection problem. Once the connection has been established, then any further IOException errors would be due to the reading of the webpage data.

|  |
| --- |
| import java.net.\*;  import java.io.\*;  public class URLConnectionReaderExample {  public static void main(String[] args) {  URL wiki = null;  BufferedReader in = null;  try {  wiki = new URL("https://en.wikipedia.org/wiki/Computer\_science");  } catch(MalformedURLException e) {  System.out.println("Cannot find webpage " + wiki);  System.exit(-1);  }  try {  URLConnection aConnection = wiki.openConnection();  in = new BufferedReader(  new InputStreamReader(aConnection.getInputStream()));  }  catch (IOException e) {  System.out.println("Cannot connect to webpage " + wiki);  System.exit(-1);  }  try {  // Now read the webpage file  String lineOfWebPage;  while ((lineOfWebPage = in.readLine()) != null)  System.out.println(lineOfWebPage);  in.close(); // Close the connection to the net  } catch(IOException e) {  System.out.println("Cannot read from webpage " + wiki);  }  }  } |