A Tutorial on Socket Programming in Java

Natarajan Meghanathan
Assistant Professor of Computer Science
Jackson State University
Jackson, MS 39217, USA
Phone: 1-601-979-3661; Fax: 1-601-979-2478
E-mail: natarajan.meghanathan@jsums.edu

Abstract

We present a tutorial on socket programming in Java. This tutorial illustrates several examples on the two types of socket APIs: connectionless datagram sockets and connection-oriented stream-mode sockets. With datagram sockets, communication occurs in the form of discrete messages sent from the sender to receiver; whereas with stream-mode sockets, data is transferred using the concept of a continuous data stream flowing from a source to a destination. For both kinds of sockets, we illustrate examples for simplex (one-way) and duplex (bi-directional) communication. We also explain in detail the difference between a concurrent server and iterative server and show example programs on how to develop the two types of servers, along with an example. The last section of the tutorial describes the Multicast Socket API in Java and illustrates examples for multicast communication. We conclude by providing descriptions for several practice programming exercises that could be attempted at after reading the tutorial.

1. Introduction

Interprocess communication (IPC) is the backbone of distributed computing. Processes are runtime representations of a program. IPC refers to the ability for separate, independent processes to communicate among themselves to collaborate on a task. The following figure illustrates basic IPC:

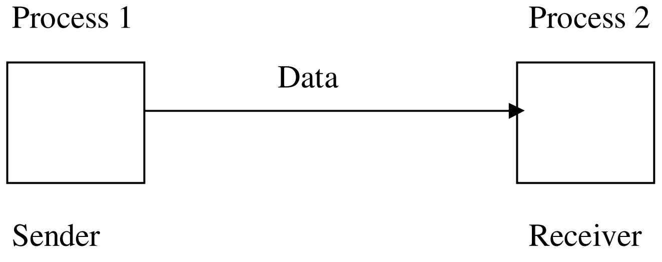

Figure 1: Interprocess Communication

Two or more processes engage in a protocol – a set of rules to be observed by the participants for data communication. A process can be a sender at some instant of the communication and can be a receiver of the data at another instant of the communication. When data is sent from one process to another single process, the communication is said to be unicast. When data is sent from one process to more than one process at the same time, the communication is said to be multicast. Note that multicast is not multiple unicasts.

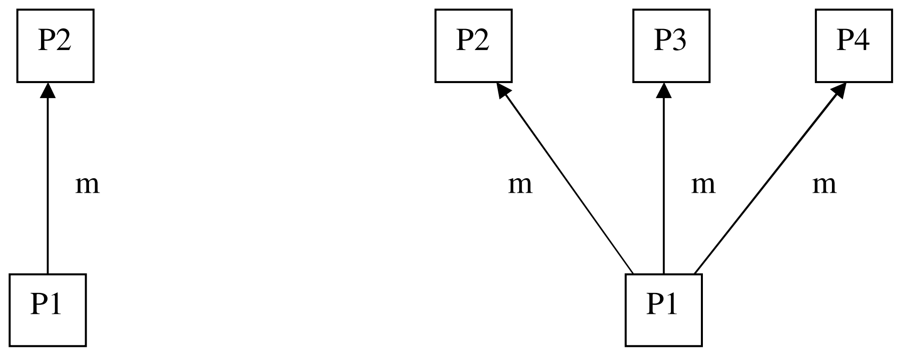

Figure 2: Unicast Vs. Multicast

Most of the operating systems (OS) like UNIX and Windows provide facilities for IPC. The system-level IPC facilities include message queues, semaphores and shared memory. It is possible to directly develop network software using these system-level facilities. Examples are network device drivers and system evaluation programs. On the other hand, the complexities of the applications require the use of some form of abstraction to spare the programmer of the system-level details. An IPC application programming interface (API) abstracts the details and intricacies of the system-level facilities and allows the programmer to concentrate on the application logic.

The Socket API is a low-level programming facility for implementing IPC. The upper-layer facilities are built on top of the operations provided by the Socket API. The Socket API was originally provided as part of the Berkeley UNIX OS, but has been later ported to all operating systems including Sun Solaris and Windows systems. The Socket API provides a programming construct called a "socket". A process wishing to communicate with another process must create an instance or instantiate a socket. Two processes wishing to communicate can instantiate sockets and then issue operations provided by the API to send and receive data. Note that in network parlance, a packet is the unit of data transmitted over the network. Each packet contains the data (payload) and some control information (header) that includes the destination address.

A socket is uniquely identified by the IP address of the machine and the port number at which the socket is opened (i.e. bound to). Port numbers are allocated 16 bits in the packet headers and thus can be at most 66535. Well-known processes like FTP, HTTP and etc., have their sockets opened on dedicated port numbers (less than or equal to 1024). Hence, sockets corresponding to user-defined processes have to be run on port numbers greater than 1024.

In this chapter, we will discuss two types of sockets – "connectionless" and "connection-oriented" for unicast communication, multicast sockets and several programming examples to illustrate different types of communication using these sockets. All of the programming examples are illustrated in Java.

2. Types of Sockets

The User Datagram Protocol (UDP) transports packets in a connectionless manner [1]. In a connectionless communication, each data packet (also called datagram) is addressed and routed individually and may arrive at the receiver in any order. For example, if process 1 on host A sends datagrams m1 and m2 successively to process 2 on host B, the datagrams may be transported on the network through different routes and may arrive at the destination in any of the two orders: m1, m2 or m2, m1.

The Transmission Control Protocol (TCP) is connection-oriented and transports a stream of data over a logical connection established between the sender and the receiver [1]. As a result, data sent from a sender to a receiver is guaranteed to be received in the order they were sent. In the above example, messages m1 and m2 are delivered to process 2 on host B in the same order they were sent from process 1 on host A.

A socket programming construct can use either UDP or TCP transport protocols. Sockets that use UDP for transport of packets are called "datagram" sockets and sockets that use TCP for transport are called "stream" sockets.

3. The Connectionless Datagram Socket

In Java, two classes are provided for the datagram socket API: (a) The DatagramSocket class for the sockets (b) The DatagramPacket class for the packets exchanged. A process wishing to send or receive data using the datagram socket API must instantiate a DatagramSocket object, which is bound to a UDP port of the machine and local to the process.

To send a datagram to another process, the sender process must instantiate a DatagramPacket object that carries the following information: (1) a reference to a byte array that contains the payload data and (2) the destination address (the host ID and port number to which the receiver process' DatagramSocket object is bound).

At the receiving process, a DatagramSocket object must be instantiated and bound to a local port – this port should correspond to the port number carried in the datagram packet of the sender. To receive datagrams sent to the socket, the receiving process must instantiate a DatagramPacket object that references a byte array and call the receive method of the DatagramSocket object, specifying as argument, a reference to the DatagramPacket object. The program flow in the sender and receiver process is illustrated in Figure 3 and the key methods used for communication using connectionless sockets are summarized in Table 1.

Sender Program

Create a DatagramSocket object and bind it to any local port;

Place the data to send in a byte array;

Create a DatagramPacket object, specifying the data array and the receiver's address;

Invoke the send method of the DatagramSocket object and pass as argument, a reference to the DatagramPacket object.

Receiver Program

Create a DatagramSocket object and bind it to a specific local port;

Create a byte array for receiving the data;

Create a DatagramPacket object, specifying the data array.

Invoke the receive method of the socket with a reference to the DatagramPacket object.

Figure 3: Program flow in the sender and receiver process (adapted from [2])

Table 1: Key Commonly Used Methods of the DatagramSocket API (adapted from [3])

No.	**Constructor/ Method**	**Description**
	DatagramSocket class	
1	DatagramSocket( )	Constructs an object of class DatagramSocket and binds the object to any available port on the local host machine
2	DatagramSocket(int port)	Constructs an object of class DatagramSocket and binds it to the specified port on the local host machine
3	DatagramSocket (int port, InetAddress addr)	Constructs an object of class DatagramSocket and binds it to the specified local address and port
4	void close( )	Closes the datagram socket
5	void connect(InetAddress address, int port)	Connects the datagram socket to the specified remote address and port number on the machine with that address
6	InetAddress getLocalAddress( )	Returns the local InetAddress to which the socket is connected.
7	int getLocalPort( )	Returns the port number on the local host to which the datagram socket is bound
8	InetAddress getInetAddress( )	Returns the IP address to which the datagram socket is connected to at the remote side.
9	int getPort( )	Returns the port number at the remote side of the socket
10	void receive(DatagramPacket packet)	Receives a datagram packet object from this socket
11	void send(DatagramPacket packet)	Sends a datagram packet object from this socket
12	void setSoTimeout(int timeout)	Set the timeout value for the socket, in milliseconds
	DatagramPacket class	
13	DatagramPacket(byte[ ] buf, int length, InetAddress, int port)	Constructs a datagram packet object with the contents stored in a byte array, buf, of specified length to a machine with the specified IP address and port number
14	InetAddress getAddress( )	Returns the IP address of the machine at the remote side to which the datagram is being sent or from which the datagram was received
15	byte [ ] getData( )	Returns the data buffer stored in the packet as a byte array
16	int getLength( )	Returns the length of the data buffer in the datagram packet sent or received
17	int getPort( )	Returns the port number to which the datagram socket is bound to which the datagram is being sent or from which the datagram is received
18	void setData(byte [ ])	Sets the data buffer for the datagram packet
19	void setAddress(InetAddress iaddr)	Sets the datagram packet with the IP address of the remote machine to which the packet is being sent
20	void setPort(int port)	Sets the datagram packet with the port number of the datagram socket at the remote host to which the packet is sent

With connectionless sockets, a DatagramSocket object bound to a process can be used to send datagrams to different destinations. Also, multiple processes can simultaneously send datagrams to the same socket bound to a receiving process. In such a situation, the order of the arrival of the datagrams may not be consistent with the order they were sent from the different processes. Note that in connection-oriented or connectionless Socket APIs, the send operations are non-blocking and the receive operations are blocking. A process continues its execution after the issuance of a *send* method call. On the other hand, once a process calls the *receive* method on a socket, the process is suspended until a datagram is received. To avoid indefinite blocking, the *setSoTimeout* method can be called on the DatagramSocket object.

We now present several sample programs to illustrate the use of the DatagramSocket and DatagramPacket API. Note that in all these exercises, the receiver programs should be started first before starting the sender program. This is analogous to the fact that in any conversation, a receiver should be tuned and willing to hear and receive the information spoken (sent) by the sender. If the receiver is not turned on, then whatever the message was sent will be dropped at the receiving side. The following code segments illustrate the code to send to a datagram packet from one host IP address and port number and receive the same packet at another IP address and port number. Though the sender and receiver programs are normally run at two different hosts, sometimes one can test the correctness of their code by running the two programs on the same host using '*localhost*' as the name of the host at remote side. This is the approach we use in this book chapter. For all socket programs, the package java.net should be imported; and very often we need to also import the java.io package to do any input/output with the sockets. Of course, for any file access, we also need to import the java.io package. Also, since many of the methods (for both the Connectionless and Stream-mode API) could raise exceptions, it is recommended to put the entire code inside a *try-catch* block.

3.1 *Example Program to Send and Receive a Message using Connectionless Sockets*

The datagram receiver (datagramReceiver.java) program illustrated below can receive a datagram packet of size at most 40 bytes. As explained before, the receive( ) method call on the

---

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

class datagramReceiver{
  public static void main(String[ ] args){
    try{
      int MAX_LEN = 40;
      int localPortNum = Integer.parseInt(args[0]);
      DatagramSocket mySocket  = new DatagramSocket(localPortNum);
      byte[] buffer = new byte[MAX_LEN];
      DatagramPacket packet = new DatagramPacket(buffer, MAX_LEN);
      mySocket.receive(packet);
      String message = new String(buffer);
      System.out.println(message);
      mySocket.close( );
     }
    catch(Exception e){e.printStackTrace( );}
   }
}
```

Figure 4: Program to Receive a Single Datagram Packet

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