Windows and Java socket programming

Stefan D. Bruda

CS 464/564, Fall 2023

Windows socket programming

- Most (but not all) declarations are in one header `winsock2.h`
- Your socket programs must link to the Winsock library `ws2_32.lib`
- Typical preamble:

```
// Only if we need windows.h:
#ifndef WIN32_LEAN_AND_MEAN
#define WIN32_LEAN_AND_MEAN
#endif

#include <windows.h>   // Optional, most of the time not needed
#include <winsock2.h>
#include <ws2tcpip.h>
#include <iphlpapi.h>
#include <stdio.h>

#pragma comment(lib, "Ws2_32.lib")
```


Windows sockets: Initialization

- Required initialization:

```
WSADATA wsaData;
// we use winsock version 2.2
int status = WSAStartup(MAKEWORD(2,2), &wsaData);
if (status != 0) {
    fprintf(stderr, "WSAStartup failed: %d\n", status);
    return 1;
}
```

- Rest of the API slightly different (but identical in functionality)
  - Not integrated in an overall OS API like in POSIX
  - For example, no integrated mechanism for reporting errors (the socket API will not set `errno`)

Windows sockets: Client socket

```
struct addrinfo *result = NULL, *ptr = NULL, hints;

ZeroMemory(&hints, sizeof(hints));
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;

// Resolve the server address and port number
int status = getaddrinfo(argv[1], port, &hints, &result);
//Check status, exit (with WSACleanup) on failure

SOCKET sd = INVALID_SOCKET;
ptr=result;
sd = socket(ptr->ai_family, ptr->ai_socktype, ptr->ai_protocol);
if (ConnectSocket == INVALID_SOCKET) {
    printf("Error at socket(): %ld\n", WSAGetLastError());
    freeaddrinfo(result);
    WSACleanup();
    return 1;
}

status = connect(sd, ptr->ai_addr, (int)ptr->ai_addrlen);
if (iResult == SOCKET_ERROR) {
    closesocket(sd);
    ConnectSocket = INVALID_SOCKET;
}
```

Windows sockets: Server socket

```
struct addrinfo *result = NULL, *ptr = NULL, hints;

ZeroMemory(&hints, sizeof (hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
hints.ai_flags = AI_PASSIVE;

status = getaddrinfo(NULL, DEFAULT_PORT, &hints, &result);
// Error if status != 0 as on the client side

SOCKET sd = INVALID_SOCKET;
sd = socket(result->ai_family, result->ai_socktype, result->ai_protocol);
// Error if sd == INVALID_SOCKET as on the client side

status = bind(sd, result->ai_addr, (int)result->ai_addrlen);
if (iResult == SOCKET_ERROR) {
    freeaddrinfo(result);
    closesocket(ListenSocket);
    WSACleanup();
    return 1;
}

if (listen(ListenSocket, backlog) == SOCKET_ERROR)
    // Error handling as above
```


Windows sockets: Communication

- On the server side use `accept()` as before to obtain the slave socket
  - On failure `accept()` returns `INVALID_SOCKET`
- Use `send()` and `recv()` as usual
  - `read()` and `write()` are not available
- `poll` (`select`, etc) only work on socket descriptors
  - They do not work on file descriptors in general
  - In particular they do not work on descriptors 0, 1, or 2
- The non-socket API is similar in function, but there are notable differences
  - In particular, `fork()` does not exist in Windows
  - Creating processes is expensive and so concurrent applications should use threads
  - In general Windows in not a POSIX compliant system, unless you find a POSIX library
  - the C standard libraries are available, POSIX calls may not be present

Windows sockets: Cleanup

- `shutdown()` available and recommended
  - Shutting down a socket for writing actually release resources
- Must use `closesocket()` instead of `close()`
- Must call `WSACleanup()` to deallocate resources
- `freeaddrinfo()` deallocates address information

Java

- Object wrappers to socket system calls
  - No access to raw sockets
- Java's thread system for concurrent applications (no `fork`)
  - Daemonization not applicable (except at the JVM level)
  - In particular, no redirection of I/O streams is possible
    - one should log to files explicitly
- Very few daemon written in Java
  - Java may of course come in handy for non-daemon distributed applications

JAVA: COMMUNICATION SOCKET

```
import java.io.DataInputStream;      import java.io.IOException;
import java.io.PrintStream;          import java.net.Socket;

class CommSocket {
    private DataInputStream input;   private PrintStream output;

    public CommSocket(Socket s) throws IOException {
        input=new DataInputStream(s.getInputStream());
        output=new PrintStream(s.getOutputStream());   }
    public void send (String msg){  output.println(msg+"\r");   }
    public void close () {
        try {  socket.close();  }
        catch (IOException e) { e.printStackTrace(); }  }
    public String receive() {
        String msgBuffer = null;
        try {  if (input.available() > 0) msgBuffer = input.readLine();  }
        catch (IOException e) {  e.printStackTrace();  }
        return msgBuffer;    }
    public String blockReceive(){
        String buffer = null;
        try {  buffer = input.readLine();  }
        catch (IOException e) {  e.printStackTrace();  }
        return buffer;    }
}
```


JAVA: SERVER

```
import java.io.IOException;
import java.net.ServerSocket;      import java.net.Socket;

public class Server implements Runnable {
    private ServerSocket socket;

    public Server(int port) {
        try {  socket = new ServerSocket(port);  }
        catch (IOException e) {  e.printStackTrace(); socket = null;  }   }
    public void run() {
        Socket incoming = null;
        while (socket != null) {
            try {  incoming  = socket.accept();
                   System.out.println("New client");
                   ClientHandler x = new ClientHandler(incoming);  }
            catch (IOException e) {  e.printStackTrace();  }
            (Thread.currentThread()).yield();
        }   }
    public static void main(String argv[]) {
        Server server = new Server(9000);
        System.out.println("Server up");
        server.run();   }
}
```


JAVA: CLIENT HANDLER

```
import java.net.Socket;
import java.io.IOException;

class ClientHandler extends Thread {
    private Socket socket;
    private CommSocket clientComm;

    public ClientHandler(Socket s) {
        socket = s;
        try {  clientComm = new CommSocket (socket);  }
        catch (IOException e) {  e.printStackTrace();  }
        this.start();    }
    public void run() {
        String s = null;
        while(true) {
            s = clientComm.blockReceive();
            if (s == null) {  System.out.println("Peer closed connection");
                              clientComm.close(); return; }
            if (s.equals("QUIT")) {  System.out.println("Connection closed");
                                     clientComm.close(); return; }
            clientComm.send("ACK: " + s);
        }    }
}
```


JAVA: MUTUAL EXCLUSION

- The only mutual exclusion mechanism: access to an object may be made mutually exclusive by using the `synchronized` statement

  `synchronized (object) { statements }`

  `synchronized (counter) { counter.increment(); }`
- A method can be declared `synchronized` as well, case in which the body of the method runs in a critical region

```
public class Placeholder {
    private int contents;
    public int get() {
        return contents;   }
    public synchronized void put(int value) {
        contents = value;   }
}
```

  - To ensure exclusive access to an object all object methods should be declared using the `synchronized` modifier