```
// VendMux.java - VENDING MACHINE MULTIPLEXER
//
// MODULE INDEX
// NAME                                 CONTENTS
// VendMux                              Vending machine multiplexor class
// VendMux.MessageProcessor             Message processor class
// VendMux.MessageProcessor.run         Message processor main line
// VendMux.MessageProcessor.mepSend     Send a message to the message processor
// VendMux.MessageProcessor.mepStart    Start the message processor
// VendMux.MessageProcessor.mepStop     Terminate the message processor
// VendMux.MessageProcessor.MessageProcessor    Construct a message processor
// VendMux.Unpack                       Packer unpacker class
// VendMux.Unpack.unpackNumber          Unpack a number
// VendMux.Unpack.Unpack                Construct packet unpacker
// VendMux.Pack                         Packet packer class
// VendMux.Pack.packNumber              Pack a number
// VendMux.Pack.Pack                    Construct a packer packer
// VendMux.muxCloseSocket               Close a socket and associated streams
// VendMux.ClientIf                     Client interface class
// VendMux.ClientIf.cifSendFault        Send a fault message
// VendMux.ClientIf.cifRecvAndRelay     Receive and relay a request
// VendMux.ClientIf.mepCall             Process a message
// VendMux.ClientIf.ClientIf            Construct a client interface
// VendMux.ServerIf                     Server interface class
// VendMux.ServerIf.sifSaveState        Save the server interface state
// VendMux.ServerIf.sifReadState        Read the server interface state
// VendMux.ServerIf.sifConnect          Connect to the vending machine
// VendMux.ServerIf.sifSubmitReq        Submit a request to the vending machine
//                                      and receive a response
// VendMux.ServerIf.mepCall             Process a message
// VendMux.ServerIf.ServerIf            Construct a server interface
// VendMux.VendMux                      Construct a vending machine multiplexer
// VendMux.main                         Main line
//
// MAINTENANCE HISTORY
// DATE         PROGRAMMER AND DETAILS
// 03-09-05     JS      Original
// 06-09-05     LSH     Renumber client packets received before sending to host
//
//-----------------------------------------------------------------------------

// IMPORTATIONS

import java.io.*;
import java.util.*;
import java.net.*;

//-----------------------------------------------------------------------------

// VENDING MACHINE MULTIPLEXOR CLASS

class VendMux
{
    //-------------------------------------------------------------------------

    // DEFINITIONS

    static final String VEND_NAME = "127.0.0.1";
                                        // Vending machine's host name
    static final int    VEND_PORT = 2251;
                                        // Vending machine's port number
    static final int    MUX_LISTEN_PORT = 2252;
                                        // Multiplexor's listening port number
    static final int    MUX_CONNECT_PORT = 2253;
                                        // Multiplexor's connecting port number
    static final int    COIN_PORT = 2254;
                                        // Coin acceptor's connecting port
    static final int    E_PURSE_PORT = 2255;
                                        // E-purse i/f's connecting port
    static final int    REQUEST_LEN = 8;
                                        // Request message length
    static final int    RESPONSE_LEN = 8;
                                        // Response message length
    static final int [] CLIENT_PORT_ARR = { COIN_PORT, E_PURSE_PORT };
                                        // Client port number array

    //-------------------------------------------------------------------------

    // MESSAGE TYPE CODES
```

```
static final int    MSG_ACCEPT = 0;         // Accepted connection
static final int    MSG_REQUEST = 1;        // Request from client
static final int    MSG_RESPONSE = 2;       // Response from vending machine
static final int    MSG_FAULT = 3;          // Connection fault
static final int    MSG_EXIT = 4;           // Request to exit

//---------------------------------------------------------------------------

// MESSAGE CLASS

static class Message {
    int             msgType;                // Message type code
    Socket          msgSocket;              // Accepted socket
    ClientIf        msgClientIf;            // Reference to client i/f
    long            msgSerialNo;            // Message serial number
    long            msgAvailable;           // Available funds
    long            msgConsumed;            // Consumed funds
}

//---------------------------------------------------------------------------

// MESSAGE PROCESSOR CLASS

abstract static class MessageProcessor
implements Runnable
{
    //-----------------------------------------------------------------------

    // CLASS INSTANCE VARIABLES

    private String          mepName;        // Name of this msg processor
    private LinkedList      mepMeq;         // Message queue
    protected Thread        mepThread;      // Thread instance
    private boolean         mepWaiting;     // Waiting flag

    //-----------------------------------------------------------------------

    // CALL THE APPROPRIATE MESSAGE PROCESSING MODULE

    protected abstract void
    mepCall (
        Message             message)        // Message instance
    throws InterruptedException;

    //-----------------------------------------------------------------------

    // MAIN LINE

    public void
    run ()
    {
        Message             message;        // Message

        try {
            for (;;) {
                synchronized (mepMeq) {
                    try {
                        mepWaiting = true;
                        while (mepMeq.size() == 0) mepMeq.wait ();
                        message = (Message) mepMeq.removeFirst ();
                        mepWaiting = false;
                    }
                    catch (InterruptedException e) {
                        mepWaiting = false;
                        throw e;
                    }
                }
                if (message.msgType == MSG_EXIT) break;
                mepCall (message);
            }
        }
        catch (InterruptedException e) {
            // Empty
        }
        catch (RuntimeException e) {
            System.err.println ("MessageProcessor(" + mepName + ")::run: "
                + e.toString());
        }
```

```
    }

    //------------------------------------------------------------------------

    // SEND A MESSAGE TO THE MESSAGE PROCESSOR

    public void
    mepSend (
        Message             message)        // Message to send
    {
        synchronized (mepMeq) {
            mepMeq.addLast (message);
            if (mepWaiting) mepMeq.notify ();
        }
    }

    //------------------------------------------------------------------------

    // INITIATE THE MESSAGE PROCESSOR

    public void
    mepStart ()
    {
        if (mepThread == null) {
            mepThread = new Thread (this, mepName);
            mepThread.start ();
        }
    }

    //------------------------------------------------------------------------

    // TERMINATE THE MESSAGE PROCESSOR

    public void
    mepStop ()
    {
        Message             message;        // Message reference

        if (mepThread != null) {
            message = new Message ();
            message.msgType = MSG_EXIT;
            mepSend (message);
            try {
                mepThread.join ();
            }
            catch (InterruptedException e) {
                throw new RuntimeException ("Unexpected interrupt");
            }
            mepThread = null;
        }
    }

    //------------------------------------------------------------------------

    // CONSTRUCT A MESSAGE PROCESSOR

    MessageProcessor (
        String              name)
    {
        mepName = name;
        mepWaiting = false;
        mepMeq = new LinkedList ();
    }
}

//----------------------------------------------------------------------------

// PACKET UNPACKER CLASS

static class Unpack
{
    //------------------------------------------------------------------------

    // CLASS INSTANCE VARIABLES

    private byte [] upkBuf;         // Unpacking buffer
    private int     upkOfs;         // Unpacking offset

    //------------------------------------------------------------------------
```

```
    // UNPACK A NUMBER

    public long
    unpackNumber (
        int                 width)          // Number field width
    {
        int                 i;              // General purpose index
        long                n;              // The number

        n = 0;
        for (i = 0; i < width; i++)
            n = (n << 8) | (upkBuf[upkOfs++] & 0xff);
        return n;
    }

    //------------------------------------------------------------------------

    // CONSTRUCT A PACKET UNPACKER

    Unpack (
        byte []             buf)            // Source buffer
    {
        upkBuf = buf;
        upkOfs = 0;
    }
}

//----------------------------------------------------------------------------

// PACKET PACKER CLASS

static class Pack
{
    //------------------------------------------------------------------------

    // CLASS INSTANCE VARIABLES

    public byte []  packBuf;        // Packing buffer
    public int      packLen;        // Packed length

    //------------------------------------------------------------------------

    // UNPACK A NUMBER

    void
    packNumber (
        long                n,              // The number to put
        int                 width)          // Number field width
    {
        int                 i;              // General purpose index

        for (i = 0; i < width; i++)
            packBuf[packLen++] = (byte)((n >> ((width-i-1)*8)) & 0xff);
    }

    //------------------------------------------------------------------------

    // CONSTRUCT A PACKET PACKER

    Pack (
        int                 capacity)       // Buffer capacity
    {
        packBuf = new byte [ capacity ];
        packLen = 0;
    }
}

//----------------------------------------------------------------------------

// CLOSE A SOCKET AND ASSOCIATED STREAMS

public static void
muxCloseSocket (
    Socket          socket,         // Socket instance
    InputStream     inputStream,    // Input stream instance
    OutputStream    outputStream)   // Output stream instance
{
    if (outputStream != null) {
```

```
        try {
            outputStream.close ();
        } catch (IOException e) {
            // Empty
        }
    }
    if (inputStream != null) {
        try {
            inputStream.close ();
        } catch (IOException e) {
            // Empty
        }
    }
    if (socket != null) {
        try {
            socket.close ();
        } catch (IOException e) {
            // Empty
        }
    }
}

//---------------------------------------------------------------------------

// CLIENT INTERFACE CLASS

static class ClientIf
extends MessageProcessor
{
    //---------------------------------------------------------------------------

    // CLASS INSTANCE VARIABLES

    ServerIf        cifServerIf;    // Server interface instance
    Socket          cifSocket;      // Socket instance
    InputStream     cifInputStream; // Client input stream
    OutputStream    cifOutputStream; // Client output stream

    //---------------------------------------------------------------------------

    // SEND A FAULT MESSAGE

    private void
    cifSendFault (
        IOException         e)              // I/O exception
    {
        Message             message;        // Message to send to ServerIf

        // Abandon the client connection

        muxCloseSocket (cifSocket, cifInputStream, cifOutputStream);

        // Log a warning

        System.err.println ("Warning: client fault: " + e.toString());

        // Send a fault message to the Server Interface

        message = new Message ();
        message.msgType = MSG_FAULT;
        message.msgClientIf = this;
        cifServerIf.mepSend (message);
    }

    //---------------------------------------------------------------------------

    // RECEIVE AND RELAY A REQUEST

    private void
    cifRecvAndRelay ()
    {
        int                 ofs;            // Offset in receive buffer
        int                 rcvdLen;        // Received length
        byte []             reqBuf;         // Request buffer
        int                 i;              // General purpose index
        Message             message;        // Message reference
        Unpack              unpack;         // Unpacker reference

        // Receive an input buffer
```

```
    try {
        reqBuf = new byte [ REQUEST_LEN ];
        ofs = 0;
        while (ofs < REQUEST_LEN) {
            rcvdLen = cifInputStream.read (reqBuf,ofs,REQUEST_LEN-ofs);
            if (rcvdLen == -1)
                throw new IOException ("end-of-file");
            ofs += rcvdLen;
        }
    }
    catch (IOException e) {
        cifSendFault (e);
        return;
    }

    // Decode the request and relay it to the Server Interface

    message = new Message ();
    message.msgType = MSG_REQUEST;
    message.msgClientIf = this;
    unpack = new Unpack (reqBuf);
    message.msgSerialNo = unpack.unpackNumber (4);
    message.msgAvailable = unpack.unpackNumber (4);
    cifServerIf.mepSend (message);
}

//---------------------------------------------------------------------

// PROCESS A MESSAGE

protected void
mepCall (
    Message             message)        // Message instance
throws InterruptedException
{
    int                 cliPort;        // Client port number
    Pack                pack;           // Packet packing instance

    // Process the various types of message

    switch (message.msgType) {

    // Accepted connections

    case MSG_ACCEPT:

        // Save the socket in the class instance variables

        cifSocket = message.msgSocket;
        cifInputStream = null;
        cifOutputStream = null;
        try {
            cliPort = cifSocket.getPort();
            cifInputStream = cifSocket.getInputStream();
            cifOutputStream = cifSocket.getOutputStream();
        }
        catch (IOException e) {
            cifSendFault (e);
            break;
        }

        // Receive and relay a request

        cifRecvAndRelay ();
        break;

    // Process a response

    case MSG_RESPONSE:

        // Pack and send the response

        pack = new Pack (RESPONSE_LEN);
        pack.packNumber (message.msgSerialNo, 4);
        pack.packNumber (message.msgConsumed, 4);
        try {
            cifOutputStream.write (pack.packBuf, 0, pack.packLen);
        }
```

```
            // If a communications failure occurs while sending the
            // response, send a fault message to the Server Interface

            catch (IOException e) {
                cifSendFault (e);
                break;
            }

            // Receive and relay another request

            cifRecvAndRelay ();
            break;
        }
    }

    //-------------------------------------------------------------------------

    // CONSTRUCT A CLIENT INTERFACE

    ClientIf (
        String              name,           // Client name
        ServerIf            serverIf)       // Server interface
    {
        super (name);
        cifServerIf = serverIf;
    }
}

//-----------------------------------------------------------------------------

// SERVER INTERFACE CLASS

static class ServerIf
extends MessageProcessor
{
    //-------------------------------------------------------------------------

    // CLIENT STATUS STRUCTURE STATE

    private static final int CST_UNIN = 0;
                                    // Uninitialised
    private static final int CST_PROC = 1;
                                    // A request is being processed
    private static final int CST_DONE = 2;
                                    // A request is finished

    //-------------------------------------------------------------------------

    // CLIENT STATUS STRUCTURE

    class CliStatus {
        int         cstState;       // Client status structure state
        long        cstSerialNo;    // Last request serial number
        long        cstAvailable;   // Last request available amount
        long        cstConsumed;    // Last response consumed amount
    }

    //-------------------------------------------------------------------------

    // CLASS INSTANCE VARIABLES

    private Socket          sifSocket;
                                    // Socket connected to vending machine
    InputStream             sifInputStream;
                                    // Server input stream
    OutputStream            sifOutputStream;
                                    // Server output stream
    private int             sifNextCliId;
                                    // Next client identifer
    private FailsafeStorage sifFailsafeStorage;
                                    // Failsafe storage inst
    private long            sifLastSerialNo;
                                    // Last serial no sent to machine
    private CliStatus []    sifCliStatusArr;
                                    // Client status array
    private ClientIf []     sifClientIfArr;
                                    // Client interface array
```

```
//-----------------------------------------------------------------------

// SAVE THE SERVER INTERFACE STATE

void
sifSaveState ()
{
    int                 i, j;           // General purpose index
    CliStatus           cliStatus;      // Client status instance
    Pack                pack;           // Message packing instance

    pack = new Pack (128);
    pack.packNumber (sifLastSerialNo, 4);
    for (i = 0; i < CLIENT_PORT_ARR.length; i++) {
        cliStatus = sifCliStatusArr[i];
        pack.packNumber (cliStatus.cstState, 1);
        pack.packNumber (cliStatus.cstSerialNo, 4);
        pack.packNumber (cliStatus.cstConsumed, 4);
        pack.packNumber (cliStatus.cstAvailable, 4);
    }
    try {
        sifFailsafeStorage.write (pack.packBuf, pack.packLen);
    }
    catch (IOException e) {
        throw new RuntimeException ("write failsafe storage: "
            + e.toString());
    }
}

//-----------------------------------------------------------------------

// READ THE CLIENT STATUS FROM FAILSAFE STORAGE

void
sifReadState ()
{
    byte []             buf;            // Restore buffer
    int                 i;              // General purpose index
    CliStatus           cliStatus;      // Client status instance
    Unpack              unpack;         // Unpacker instance

    // Allocate client status array storage

    sifCliStatusArr = new CliStatus [ CLIENT_PORT_ARR.length ];
    for (i = 0; i < CLIENT_PORT_ARR.length; i++)
        sifCliStatusArr[i] = new CliStatus ();

    // Attempt to read the client status data from the
    // failsafe storage.  If the client status data cannot
    // be read, initialise it.

    try {
        buf = sifFailsafeStorage.read ();
    }
    catch (IOException e) {
        System.err.println ("Warning: cannot read failsafe storage: "
            + e.toString());
        sifLastSerialNo = 0;
        for (i = 0; i < CLIENT_PORT_ARR.length; i++)
            sifCliStatusArr[i].cstState = CST_UNIN;
        return;
    }

    // Unpack the client status data

    unpack = new Unpack (buf);
    sifLastSerialNo = unpack.unpackNumber (4);
    for (i = 0; i < CLIENT_PORT_ARR.length; i++) {
        cliStatus = sifCliStatusArr[i];
        cliStatus.cstState = (int) unpack.unpackNumber (1);
        cliStatus.cstSerialNo = unpack.unpackNumber (4);
        cliStatus.cstConsumed = unpack.unpackNumber (4);
        cliStatus.cstAvailable = unpack.unpackNumber (4);
    }
}

//-----------------------------------------------------------------------

// CONNECT TO THE VENDING MACHINE
```

```
void
sifConnect ()
{
    // Loop until a connection is created

    do {
        // Attempt to create a connection

        sifSocket = null;
        sifInputStream = null;
        sifOutputStream = null;
        try {
            sifSocket = new Socket (
                InetAddress.getByName (VEND_NAME),
                VEND_PORT,
                InetAddress.getLocalHost (),
                MUX_CONNECT_PORT
            );
            sifInputStream = sifSocket.getInputStream ();
            sifOutputStream = sifSocket.getOutputStream ();
        }
        catch (IOException e) {
            System.err.println ("Warning: connect to vending machine: "
                + e.toString());
            if (sifSocket != null) {
            }
            try {
                Thread.currentThread().sleep (10000);
            }
            catch (InterruptedException e2) {
                // Empty
            }
        }
    } while (sifSocket == null);
}

//---------------------------------------------------------------------

// SUBMIT A REQUEST TO THE VENDING MACHINE AND RECEIVE A RESPONSE

void
sifSubmitReq (
    CliStatus           cliStatus)      // Client status instance
{
    Pack                pack;           // Packet packing instance
    Unpack              unpack;         // Packet unpacking instance
    byte []             respBuf;        // Response buffer
    int                 ofs;            // Offset in the resonse buffer
    int                 rcvdLen;        // Received length

    // Pack the request

    pack = new Pack (REQUEST_LEN);
    pack.packNumber (sifLastSerialNo, 4);
    pack.packNumber (cliStatus.cstAvailable, 4);

    // Send the request until a response is satisfactorily received

    do {
        // If the connection to the vending machine is unserviceable,
        // create a new connection.

        if (sifSocket == null) sifConnect ();

        // Catch I/O exceptions

        try {
            // Send the request to the vending machine

            sifOutputStream.write (pack.packBuf, 0, pack.packLen);

            // Receive a response from the vending machine

            respBuf = new byte [ RESPONSE_LEN ];
            ofs = 0;
            while (ofs < RESPONSE_LEN) {
                rcvdLen = sifInputStream.read (
                        respBuf, ofs, RESPONSE_LEN-ofs);
```

```
                if (rcvdLen == -1)
                    throw new IOException ("end-of-file");
                ofs += rcvdLen;
            }

            // Decode the response

            unpack = new Unpack (respBuf);
            if (sifLastSerialNo != unpack.unpackNumber (4))
                throw new IOException ("serial number mismatch");
            cliStatus.cstConsumed = unpack.unpackNumber (4);
            cliStatus.cstState = CST_DONE;

            // Save the Server Interface state

            sifSaveState ();
        }

        // If an I/O exception is detected, abandon the socket
        // and its associated streams.

        catch (IOException e) {
            muxCloseSocket (sifSocket, sifInputStream,
                sifOutputStream);
            sifSocket = null;
        }

        // Repeat the loop until the exchange is satisfactorily
        // completed.

    } while (sifSocket == null);
}

//---------------------------------------------------------------------

// PROCESS A MESSAGE

protected void
mepCall (
    Message             message)        // Message instance
throws InterruptedException
{
    ClientIf            clientIf;       // Client interface instance
    CliStatus           cliStatus;      // Client status instance
    Message             newMessage;     // New message instance
    int                 i;              // General purpose index
    int                 cliPort;        // Client port
    int                 clientNo;       // Client number

    // Process the various types of message

    switch (message.msgType) {

    // Accepted connections

    case MSG_ACCEPT:

        // Identify whether the accepted connection comes from
        // the coin acceptor or the e-purse interface

        cliPort = message.msgSocket.getPort();
        clientNo = 0;
        while (
            clientNo < CLIENT_PORT_ARR.length &&
            CLIENT_PORT_ARR[clientNo] != cliPort
        ) clientNo ++ ;

        // If the client port is not recognised, abandon the
        // connection and close the socket.

        if (clientNo >= CLIENT_PORT_ARR.length) {
            try {
                message.msgSocket.close ();
            } catch (IOException e) {
                // Empty
            }
            break;
        }
```

```
    // Create a new client interface

    clientIf = new ClientIf ("ClientIf_" + sifNextCliId++, this);
    sifClientIfArr[clientNo] = clientIf;
    clientIf.mepStart ();

    // Relay the accepted connection to the client interface

    clientIf.mepSend (message);
    break;

// Process a client request

case MSG_REQUEST:

    // Identify the client

    clientNo = 0;
    while (
        clientNo < CLIENT_PORT_ARR.length &&
        sifClientIfArr[clientNo] != message.msgClientIf
    ) clientNo ++ ;

    // If the client is not recognised, assume it is from
    // an abandoned connection and stop the client interface.

    if (clientNo >= CLIENT_PORT_ARR.length) {
        message.msgClientIf.mepStop ();
        break;
    }

    // If this is a new client request, submit the new
    // request to the server

    cliStatus = sifCliStatusArr[clientNo];
    if (
        cliStatus.cstState != CST_DONE ||
        message.msgSerialNo != cliStatus.cstSerialNo
    ) {
        // Update the client status instance

        cliStatus.cstState = CST_PROC;
        cliStatus.cstSerialNo = message.msgSerialNo;
        cliStatus.cstAvailable = message.msgAvailable;

        // Allocate a new serial number for the server

        sifLastSerialNo = (sifLastSerialNo + 1) & 0xffffffffL;

        // Save the Server Interface state

        sifSaveState ();

        // Submit the request to the vending machine

        sifSubmitReq (cliStatus);
    }

    // Return the response to the vending machine

    newMessage = new Message ();
    newMessage.msgType = MSG_RESPONSE;
    newMessage.msgSerialNo = cliStatus.cstSerialNo;
    newMessage.msgConsumed = cliStatus.cstConsumed;
    message.msgClientIf.mepSend (newMessage);
    break;

// Process a client fault

case MSG_FAULT:

    // Identify the client

    clientNo = 0;
    while (
        clientNo < CLIENT_PORT_ARR.length &&
        sifClientIfArr[clientNo] != message.msgClientIf
    ) clientNo ++ ;
```

```
            // If the client is the current client for this port
            // reset the reference to the client interface

            if (clientNo < CLIENT_PORT_ARR.length)
                sifClientIfArr[clientNo] = null;

            // Stop the client interface

            message.msgClientIf.mepStop ();
            break;
        }
    }

    //-------------------------------------------------------------------------

    // CONSTRUCT A SERVER INTERFACE

    ServerIf (
        String              name)           // Client name
    {
        // Construct the super-class

        super (name);

        int                 i;              // General purpose index

        // Initialise the class instance variables

        sifSocket = null;
        sifNextCliId = 0;
        sifFailsafeStorage = new FailsafeStorage ();
        sifClientIfArr = new ClientIf [ CLIENT_PORT_ARR.length ];

        // Retrieve the last saved server interface state

        sifReadState ();

        // Connect to the vending machine

        sifConnect ();

        // If a client request was in progress, resubmit the
        // request

        for (i = 0; i < CLIENT_PORT_ARR.length; i++)
            if (sifCliStatusArr[i].cstState == CST_PROC)
                sifSubmitReq (sifCliStatusArr[i]);
    }
}

//-----------------------------------------------------------------------------

// CONSTRUCT A VENDING MACHINE MULTIPLEXER

VendMux ()
{
    ServerIf        serverIf;       // Server interface
    ServerSocket    acceptSocket;   // Socket to accept client connections
    Message         message;        // A message

    // Initiate the Server Interface

    serverIf = new ServerIf ("ServerIf");
    serverIf.mepStart ();

    // Open a socket to accept connections from the clients

    acceptSocket = null;
    try {
        acceptSocket = new ServerSocket (MUX_LISTEN_PORT);
    }
    catch (IOException e) {
        throw new RuntimeException ("open listen port: " + e.toString());
    }

    // Loop to accept connections

    for (;;) {
```

```
            // Accept a connection from the listening socket

            try {
                message = new Message ();
                message.msgType = MSG_ACCEPT;
                message.msgSocket = acceptSocket.accept ();
                serverIf.mepSend (message);
            }
            catch (IOException e) {
                System.err.println ("Warning: accept socket: " + e.toString());
            }
        }
    }

    //---------------------------------------------------------------------------

    // MAIN LINE

    static public void
    main (
        String []       argv)           // Argument values
    {
        new VendMux ();
    }
}
```