LIVECODE (https://livecode.com) Platform (https://livecode.com/products/livecode-platform/) Resources (https://livecode.com/resources/)

Pricing (https://livecode.com/products/livecode-platform/pricing/) Services (https://livecode.com/services/) Blog (https://livecode.com/blog/)

login (https://livecode.com/login/)

Dictionary (https://livecode.com/resources/api/) Guides (https://livecode.com/resources/guides/) Lessons (https://lessons.livecode.com/) Courses (https://livecode.com/products/learn/)

How to communicate with other applications using sockets

Sockets provide the programmer with a facility to allow their applications to communicate over a network. This lesson shows how to use sockets to allow to LiveCode apps to talk to each other as well as providing an example of how LiveCode apps can share data with Java apps.

What are Sockets?

Sockets provide two networked machines with a bidirectional communication channel. One machine will be the server, listening in for connections and the other, the client, attempting to make a connection with the server. An example of this would be when you fetch a web page. Your web browser, or the client, attempts to make a connection to the web server. The web server, listening in for clients, will accept the connection and then proceed to handle the clients request. Here, the browser will ask the web server for the page, before the web server responds with the page data.

Servers are accessed via socket addresses, a combination of the server's IP address (or domain name) and a port number. The port can be thought as a connection point on the sever, like USB or Firewire ports, with each port serving a specific purpose. For example, web pages are served on port 80 (HTTP) , emails are sent via port 25 (SMTP).

Once two machines are connected, they can then communicate streams of bytes with each other. It's up to the client and server to format these byte streams into structured data chunks. A simple example would be an echo sever, which receives a stream of bytes from a client, assumes they form an ascii string and sends the string back to the client.

As noted previously, sockets use IP based networks. There are a range of transport layers that can be used on top of IP. In this lesson we will consider TCP sockets.

How to Use Sockets in LiveCode - Server Side

Using sockets in LiveCode is very simple. First of all let us consider a server: A server must listen in on a specific port for clients. To do this in LiveCode, we simply use the following command:

```
accept connections on port 1234 with message "clientConnected"
```

Here, our server will listen in on port 1234 for connecting clients. When a client is connected, the message "clientConnected" will be sent, detailing the newly opened socket. The server can now listen in on this socket for messages from the client:

```
on clientConnected pSocket
       read from socket pSocket with message "messageReceived"
end clientConnected
```

In the example above, the server is reading from the client in non-blocking mode. That means that the read request will exit instantly. When a data is received from the client, the "messageReceived" message will be sent, detailing the socket it came from and the data sent. The alternative to this is to omit the "with message". Here the read will function in blocking mode, not returning until data has been received from the client. When data is received, the read command will return and the data will be placed in it.

An additional parameter can be passed to the read command, detailing how much data we want to read. This can be a number of characters or lines (or any other chunk type - read from socket pSocket for 5 lines) or until a specific character is reached (read from socket pSocket until return).

Let us consider the read as detailed earlier, in non-blocking mode. When data is received from the client the "messageReceived" message will be sent. At this point, the server can choose to write a response to the client. Here, we will just echo the data back to the client using the write command.

```
on messageReceived pSocket, pMsg
        write pMsg to socket pSocket
        read from socket pSocket with message "messageReceived"
end messageReceived
```

Like the read command, write can be blocking or non-blocking. If we want a non blocking write, we simply specify the message to be sent when the write is complete. Note that in order to continue reading from the client, we must issue another read command.

When the server no longer wishes to communicate with the client, the socket can simply be closed with the command close socket tSocket. If the client closes the socket, the server will be sent a socketClosed message, detailing the socket that has been closed.

How to User Sockets in LiveCode - Client Side

Using sockets on the client side is equally as easy in LiveCode. To begin communicating with the server, we must first open a socket:

```
open socket to "host:port" with message "clientConnected"
```

Like the read and write commands, the callback message on the open command is optional, and defines if the command is blocking or non-blocking. If used in blocking mode, once connected the newly opened socket will be accessible via the it variable. In non-blocking mode, it will be passed as a parameter in the callback message.

If there has been an error connecting to the server the "socketError" message will be sent. (If the error is due to a problem finding the specified host, the error message is returned in the result, and no "socketError" message is sent.)

Let us consider the open command in non-blocking mode. Once the socket is opened, we can write a message to the server, read the response and then close the connection:

```
on clientConnected pSocket
        write "hello" & return to socket pSocket
        read from socket pSocket until return
        put it
        close socket pSocket
end clientConnected
```

The read and write commands function in exactly the same manner as defined previously for the server.

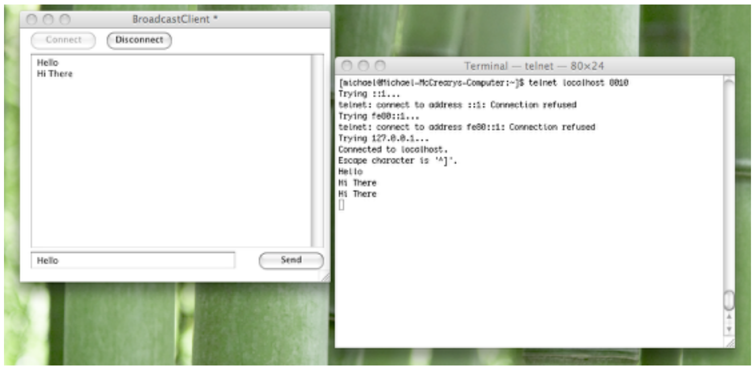
Creating a Simple Broadcast Server

Let us expand on these commands and create a simple broadcast server. Our broadcast server will listen in for connecting clients. Once a client is connected, the server will place the client on its list of connected clients before attempting to read a line of data from it. When a line of data is read from a client, that data will be broadcast to each connected client.

The list of connected clients will simply be a script variable holding the socket for each client separated by return. Our server will be started and stopped using simple start and stop commands. Two further handlers will be defined: One called when a client is connected, adding the client to the list before beginning reading. The other will be called when the read is complete, broadcasting the read message to all connected clients.

```
local sConnectedClients        -- list of authorised clients
local sRunning                                -- if the server is currently running
constant kPort = 8010

-- Start the server listening for connections.  When a client attempts to
-- connect, the broadcastServerClientConnected message will be sent.
command broadcastServerStart
       if not sRunning then
              put true into sRunning
              accept connections on port kPort \
                             with message "broadcastServerClientConnected"
       end if
end broadcastServerStart
-- Stop the server listening for connections.  Close all open communication
-- channels.

command broadcastServerStop
       if sRunning then
              put false into sRunning
              put empty into sConnectedClients
              repeat for each line tSocket in the opensockets
                     close socket tSocket
              end repeat
       end if
end broadcastServerStop

-- Sent when a client attempts to make a connection.  Store the client
-- in the connected list and begin reading lines of text from them.  The
-- broadcastServerMessageReceived message will be sent when a line is received.
on broadcastServerClientConnected pSocket
       put pSocket & return after sConnectedClients
       read from socket pSocket until return \
                     with message "broadcastServerMessageReceived"
end broadcastServerClientConnected

-- Sent when a line of text is read from a client.  Broadcast that line of
-- text to all connected clients.
on broadcastServerMessageReceived pSocket, pMsg
       repeat for each line tSocket in sConnectedClients
              write pMsg to socket tSocket
       end repeat
       read from socket pSocket until return \
                     with message "broadcastServerMessageReceived"
end broadcastServerMessageReceived

-- Sent when a client disconnects.  Remove the client from connected list
on socketClosed pSocket
       delete line lineoffset(pSocket, sConnectedClients) of sConnectedCLients
end socketClosed
```

Creating a Broadcast Client



```
Hello
Hi There
```

```
Hello
```

```
[michael@Michael-McCrearys-Computer:~]$ telnet localhost 8010
Trying ::1...
telnet: connect to address ::1: Connection refused
Trying fe80::1...
telnet: connect to address fe80::1: Connection refused
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello
Hi There
Hi There
```

Now lets define our client. This is really easy. We just, as before, connected to the server. Once connected, we can begin sending messages by calling the "sendMessage" handler. This handler will simply write the message to the server using the socket we connected with.

The other task of the client is to listen for any messages broadcast by the server. Here, the client will just read from the server in non-blocking mode, logging the message read before continuing the read.

```
local sSocket
constant kPort = 8010

-- Connect the client to the given server.  Will send a broadcastClientConnected
-- when the client has connected.
command broadcastClientStart pServer
        if sSocket is empty then
                open socket to pServer & ":" & kPort \
                                with message "broadcastClientConnected"
        end if
end broadcastClientStart

-- Disconnect the client from the broadcast server.
command broadcastClientStop
        if sSocket is not empty then
                close socket sSocket
                put empty into sSocket
        end if
end broadcastClientStop

-- Write the given message to the communication channel the client
-- has opended with the broadcast server.
command broadcastClientSend pMsg
        if sSocket is not empty then
                write pMsg & return to socket sSocket
        end if
end broadcastClientSend

-- Sent once the client has connected to the broadcaset server.
-- Store the socket for futurure reference and begin reading data
-- from the server.
on broadcastClientConnected pSocket
        put pSocket into sSocket
        read from socket sSocket until return \
                        with message "broadcaseClientMessageReceived"
end broadcastClientConnected

-- Sent when a message has been received from the server.  Output the
-- message and continue reading data from the server.
on broadcaseClientMessageReceived pSocket, pMsg
        put pMsg after field "log"
        read from socket sSocket until return \
                        with message "broadcaseClientMessageReceived"
end broadcaseClientMessageReceived

-- Sent when there is an error opening the socket.  Log the error.
-- and close the socket.
on socketError pSocket, pError
        broadcastClientDisconnect
        put pError & return after field "log"
end socketError

-- Sent when the connection to the server is closed by the server.
on socketClosed
        put empty into sSocket
end socketClosed
```

The image above shows our broadcast client communicating with the server and another client connected via telnet.

Devising a Communication Protocol

Our broadcast client/server used a very simple communication protocol, where messages were sent as ASCII text blocks separated by a return character. A communication protocol simply defines the means in which the transmitted data is encapsulated. Producing a simple and robust protocol allows your system to implemented in range of different manners, using various different technologies. Consider the number of different web browsers (and servers) on offer.

Let us develop a very simple chat protocol. First of all, each message transmitted will have a header. Our header will comprise of a 4 character message type code and an integer detailing the number of bytes (or characters) in our message body. The header will be split from the message body by a return character.

We will include 5 message types:

AUTH: Sent by the client, requesting authorization.

VERI: Returned by the server if the client's authorization request is successful.

MESG: Sent by both client and server, indicating a simple ASCII text message to be broadcast.

ERRO: Sent by the server to indicate that there has been a error (authorization failed etc) and the client will be disconnected.

WARN: Sent by the server to indicate a warning, such as the client has sent an invalid message.

Such a protocol may seem a little excessive (especially in our example) but it provides a robust programmers interface and a solid platform for future extensions. For example, we may wish to include file sharing, making the message size an important piece of information.

LiveCode Chat Server

We will implement our chat server protocol using LiveCode. Our server will function in a very similar manner to our broadcast server. This time, however, once a client is connected, the will go on the pending list, awaiting authorization. Authorization will be a simple tasks of ensuring the client has a uniques user name. Once authorized, the client will be moved on to the authorized list and be allowed to send and receive messages.

The main handler of interest will be the message received callback. Here, the server must parse the header and if required read the message body. To do this, we will use the "read from for" command, reading the number of characters defined in the message header.

```
local sConnectedClients        -- array of authorised clients [client] => [name]
local sPendingClients          -- list of all pending clients
local sClientNames                     -- list of all currently used client names
local sRunning                                 -- if the server is currently running
constant kPort = 8020

-- Start the server listening for connections.  When a client
-- attempts to connect, the chatServerClientConnected message will be sent.
command chatServerStart
        if not sRunning then
                put true into sRunning
                accept connections on port kPort \
                                with message "chatServerClientConnected"
        end if
end chatServerStart

-- Stop the server listening for connections.  Close all open communication
-- channels.
command chatServerStop
        if sRunning then
                put false into sRunning
                put empty into sConnectedClients
                put empty into sPendingClients
                put empty into sClientNames
                repeat for each line tSocket in the opensockets
                        close socket tSocket
                end repeat
        end if
end chatServerStop

-- Sent when a client attempts to make a connection.  Store the client as
-- pending and begin reading message headers (line of text) from the client.
-- When a header is received, a chatServerMessageReceived message will be sent.
on chatServerClientConnected pSocket
       put pSocket & return after sPendingClients
       read from socket pSocket until return \
                       with message "chatServerMessageReceived"
end chatServerClientConnected

-- Sent when a message header is received from a client.  Parse the header and
-- the read message body if required.  Do this by switching accross the message
-- type (first item in header).
--
-- If it is an authorisation request, read the message body (clients name),
-- check it's unique, add client to list of authorised clients and send
-- authorisation success message to client.
--
-- If it is a broadcast message, check the client is authorised and then send
-- to all connected clients.
--
-- Once message is handled, continue reading from client.
on chatServerMessageReceived pSocket, pMsg
       put char 1 to -2 of pMsg into pMsg
       local tAuth, tCommand, tLength, tMsg
       put pSocket is among the keys of sConnectedClients into tAuth
       put item 1 of pMsg into tCommand
       put item 2 of pMsg into tLength
       if tLength is not an integer then
              put "Invalid message length" into tMsg
              write "WARN," & the number of chars in tMsg \
                             & return & tMsg to socket pSocket
       else
              switch tCommand
                     case "MESG"
                            if tAuth then
                                   read from socket pSocket for tLength chars
                                   chatServerBroadcast sConnectedClients[pSocket] & ":" && it
                            else
                                   put "Client not verified" into tMsg
                                   write "ERRO," & the number of chars in tMsg \
                                                 & return & tMsg to socket pSocket
                            end if
                            break
                     case "AUTH"
                            if tAuth then
                                   put "Client already verified" into tMsg
                                   write "WARN," & the number of chars in tMsg \
                                                 & return & tMsg to socket pSocket
                            else
                                   read from socket pSocket for tLength chars
                                   if it is not among the lines of sClientNames then
                                          put it into sConnectedClients[pSocket]
                                          put it & return after sClientNames
                                          write "VERI,0" & return to socket pSocket
                                          delete line \
                                                        lineoffset(pSocket, sPendingClients) of sPendingClients
                                          chatServerBroadcast it && "connected"
                                   else
                                          put "Username already taken" into tMsg
                                          write "ERRO," & the number of chars in tMsg \
                                                                & return & tMsg to socket pSocket
                                        end if
                                end if
                                break
                        default
                                put "Unknown command" into tMsg
                                write "ERRO," & the number of chars in tMsg \
                                                & return & tMsg to socket pSocket
                                break
                end switch
        end if
        read from socket pSocket until return \
                         with message "chatServerMessageReceived"
end chatServerMessageReceived

-- Broadcasts the given message to all connected and verified clients.
command chatServerBroadcast pMsg
        local tMsg
        put "MESG," & the number of chars in pMsg & return & pMsg into tMsg
        repeat for each line tSocket in the keys of sConnectedClients
                write tMsg to socket tSocket
        end repeat
end chatServerBroadcast

-- Sent when a client disconnects.  Remove the client from the pending
-- list if they are pending, or authorised list if they are authorised.
on socketClosed pSocket
        if pSocket is among the lines of sPendingClients then
                delete line lineoffset(pSocket, sPendingClients) of sPendingClients
        else if sConnectedClients[pSocket] is not empty then
                local tName
                put sConnectedClients[pSocket] into tName
                delete variable sConnectedClients[pSocket]
                delete line lineoffset(tName, sClientNames) of sClientNames
                chatServerBroadcast tName && "disconnected"
        end if
end socketClosed
```

Java Chat Client

Though we could easily develop our client in LiveCode, in this final part of the lesson we will consider Java. Doing so demonstrates how we can use sockets for inter-application communication and the importance of using a clearly defined protocol.

I won't go into to too much detail here, but will note a few key points. Since we will be working with sockets, we'll need to import the Java "net" library. Also, since we will be using input and output streams, we'll need the "io" library. In order to open a socket, we create a new object of type Socket:

```
Socket tSocket = new Socket("hostname", "port");
```

Once we have opened our socket, we can then fetch the input and output streams. These streams allow us to read and write data to and from the server. Java provides various wrappers allowing the data read and written to be formatted in various ways: At the simplest level, you may wish to read and write directly to the socket, handling data a byte at a time. Alternatively, you may wish to operate at a higher level, sending data as unicode text.

In our example we use a BufferedReader for the input stream (as it allows us to read a single line of text - our message header - as well as a fixed block of text - our message body).

```
BufferedReader tInput = new BufferedReader(new InputStreamReader(tSocket.getInputStream()));
String tHeader = tInput.readLine();
For output, we use a DataOutputStream as it allows out to write a stream of bytes - our ASCII encoded message.
DataOutputStream tOutput = new DataOutputStream(tSocket.getOutputStream());
tOutput.write("MESG,5\nHello".getBytes());
tOutput.flush();
```

Notice that after we write to the output stream, we call the flush method. This ensures that the data is sent immediately, rather than queued up in the stream.

Once the client is connected to the server, we start a new thread that listens to the input stream for data from the server. We do this by making our client class implement the Runnable interface, then filling out the run method of our client. The run method will be called once the new thread is created and must continually listen to the input stream whilst the client is connected.

All the main functionality will be wrapped up in a single Client class. Our Client class will have three main public methods available to the user: connect, disconnect and send. Note the use of an enumerative type, used to store the clients current connection status, and the ClientUI type used to handle all user interaction.

```
import java.net.*;
import java.io.*;
public enum ConnectionStatus {
        DISCONNECTED,
        CONNECTING,
        CONNECTED,
}
public class ChatClient implements Runnable {

        private ChatClientUI mUI;
        private Socket mSocket;
        private BufferedReader mInput;
        private DataOutputStream mOutput;
        private Thread mListener;
        private ConnectionStatus mStatus = ConnectionStatus.DISCONNECTED;
        private static final int kPort = 8020;
        public ChatClient(ChatClientUI pUI) {
                this.mUI = pUI;
        }


        /*
         * Connect to the given server with the  given user name.
         * Opens a communication channel with the server, sets up the input and
         * output streams, sends an authorisation request to the server
         * and starts a new thread listening for any response from the server.
         */
        public void connect(String pServer, String pUsername) {
                if (pServer.equals("")) {
                        this.mUI.displayError("Connection Error: Missing server name");
                } else if (pUsername.equals("")) {
                        this.mUI.displayError("Connection Error: Missing user name");
                } else if (this.mStatus != ConnectionStatus.DISCONNECTED) {
                        this.mUI.displayError("Connection Error: Client already connected");
                } else {
                        try {
                                this.setStatus(ConnectionStatus.CONNECTING);
                                this.mSocket = new Socket(pServer, ChatClient.kPort);
                                this.mInput = new BufferedReader(
                                                               new InputStreamReader(this.mSocket.getInputStream()));
                                this.mOutput = new DataOutputStream(this.mSocket.getOutputStream());
                                this.mListener = new Thread(this);
```

```
			this.mListener.start();
				this.writeMessage("AUTH", pUsername);
			} catch (IOException e) {
				this.mUI.displayError("Connection Error: " + e.toString());
				this.disconnect();
			}
		}
	}

	/*
	 * Attempt to disconnect from the server.  Close socket before closing the
	 * input and output streams. Updates the client's current connection status.
	 */
	public boolean disconnect() {
		if (this.mStatus != ConnectionStatus.DISCONNECTED) {
			this.setStatus(ConnectionStatus.DISCONNECTED);
			try {
				this.mSocket.close();
				this.mInput.close();
				this.mOutput.close();
			} catch (Exception e) { }
			return true;
		} else {
			this.mUI.displayError("Disconnect Error: Client not connected");
			return false;
		}
	}

	/*
	 * Send the given chat message to the server.  Will format the message with
	 * type MESG before sending to the server for broadcast to all othe clients.
	 */
	public void send(String pMsg) {
		if (this.mStatus == ConnectionStatus.CONNECTED) {
			this.writeMessage("MESG", pMsg);
		} else {
			this.mUI.displayError("Communication error: Client not verified");
		}
	}

	/*
	 * Returns the clients current connection status.
	 */
	public ConnectionStatus getStatus() {
		return this.mStatus;
	}

	/*
	 * Method defined by Runnable interface. Called once client starts listening
	 * for server messages.  While the client is connected, attempt to read
	 * and process headers (single lines of text) from the input stream.
	 */
	public void run() {
		try {
			while (this.mStatus != ConnectionStatus.DISCONNECTED) {
				this.handleResponse(this.mInput.readLine());
			}
		} catch (IOException e) {
			if (this.mStatus != ConnectionStatus.DISCONNECTED) {
				this.mUI.displayError("Communication Error: " + e.toString());
				this.disconnect();
			}
		}
	}

	/*
	 * Set the clients current connection status.
	 * Update the ui with the new status.
	 */
	private void setStatus(ConnectionStatus pStatus) {
		if (pStatus != this.mStatus) {
			this.mStatus = pStatus;
			this.mUI.statusUpdate(pStatus);
		}
	}

	/*
	 * Send the given message to the server.  Format the header based on the
	 * message type and message body length, attach the body and then write to
	 * the output stream.
	 */
	private void writeMessage(String pType, String pMsg) {
		if (this.mStatus != ConnectionStatus.DISCONNECTED) {
			try {
				String tMsg = pType + "," + pMsg.length() + "\n" + pMsg;
				this.mOutput.write(tMsg.getBytes());
				this.mOutput.flush();
			} catch (IOException e) {
				this.mUI.displayError("Communication Error: " + e.toString());
				this.disconnect();
			}
		} else {
			this.mUI.displayError("Communication Error: Client not connected");
		}
	}

	/*
	 * Parse the passed header.  The header should be a single line of text,
	 * comma split into the message type, the message length.  Based on the type
	 * parse the message, reading the body if required.
	 */
	private void handleResponse(String pMsg) {
		if (pMsg == null) {
			this.disconnect();
			this.mUI.displayMessage("Server shutdown");
		} else {
			String[] tItems = pMsg.split(",");
			String tMsg = tItems[0];
			int tLength = Integer.parseInt(tItems[1]);
			if (tMsg.equals("MESG")) {
				this.mUI.displayMessage(this.readData(tLength));
			} else if (tMsg.equals("WARN")) {
				this.mUI.displayError(this.readData(tLength));
			} else if (tMsg.equals("ERRO")) {
				this.mUI.displayError(this.readData(tLength));
				this.disconnect();
			} else if (tMsg.equals("VERI")) {
				this.setStatus(ConnectionStatus.CONNECTED);
			} else {
				this.mUI.displayError("Communication Error: Unknown message " + tMsg);
			}
		}
	}
```

```
        /*
         * Read pLength bytes froim the input channel and pasre as a string.
         * Used to read message bodies (when lenght is defined in header).
         */
        private String readData(int pLength) {
                char [] tBuffer = new char[pLength];
                try {
                        this.mInput.read(tBuffer, 0, pLength);
                        return new String(tBuffer);
                } catch (IOException e) {
                        this.mUI.displayError("Communication Error: " + e.toString());
                        this.disconnect();
                        return null;
                }
        }
}
```

12 Comments

Bruce Jorgensen Friday Apr 09 2010 at 12:09 PM

I like the lesson; I think you need a little more explanation of why you are including code example for a Java client--is it because you want to show interoperability with other platforms or because a revTalk version of the client is too difficult to write? Seems like the whole Java thing is kind of out of context here, but maybe with a bit of explanation it will become obvious why the need.

Ben Beaumont Tuesday Apr 13 2010 at 09:21 AM

Hi Bruce,

The purpose of this lesson is to demonstrate how to talk to other applications written in other languages. We produced it on request from another user who was looking to do this very task.

Warm regards,

Ben

Peter Alcibiades Thursday Apr 29 2010 at 04:02 PM

Very nice tutorial indeed. Thanks. Shows just how to do it.

Wes Whitehead Thursday May 06 2010 at 05:11 PM

I am grateful for this tutorial i am indeed going to write a socket server using rev and using a c++ game client to talk with it so its importance to see how applications written in other languages can work with a rev powered backend.
Go Rev

Sean Cole Sunday Oct 09 2011 at 08:09 PM

Could really do with seeing how to transmit bot binary and hex data over a socket.

Elanor Buchanan Monday Oct 10 2011 at 07:19 AM

Hi Sean, thanks for the comment. I have passed your request on to the team who create the lessons.

Kind regards

Elanor

Jonny Larsen Tuesday Mar 04 2014 at 04:25 PM

Hello together,
the code sounds quite easy, unfortunately I don't get it running. On the server side I have a python script running, as I use special python libraries that I need. It looks like the following:

s=socket.socker(socket.AF_INET,socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
s.bind("",55000)
s.listen(1)

The script works quite good with another python script as client, that I used for a test till now. Unfortunately I do not get it running with livecode. I used the script from above in the card code:

open socket to "host:port" with message "clientConnected"
on clientConnected pSocket
write "hello" & return to socket pSocket
read from socket sSocket until return
put it
close socket pSocket
end clientConnected

The problem is, I don't get a connection at all. Is there something that I do wrong?
Kind regards! Jonny

Ernie St. Louis Monday Sep 15 2014 at 06:49 PM

An example detailing communication with a USB device would be nice.

Trevix Saturday Jan 03 2015 at 05:30 AM

Nice example.
Would be useful to show a Livecode version of the Java Chat Client, while taking care of the communication between multiple platform and the Unicode mess (I am lost with LC7...). Is it still better to "URLEncode(base64Encode(it))" ? What about "ISOToMac" and "MacToISO" ?
Trevix

Simon Tuesday Jul 18 2017 at 02:20 PM

This is a great tutorial but I guess there is something missing and it is about a how the client will get the Host IP address. Without it the solution is worthless unless both client and server are in the same network. Am I right? If answer is yes any hints as how to do it?

Giorgio Thursday Aug 02 2018 at 05:27 AM

Yes I also would be interested how the client get the Host IP. And I would like to connect from mobile phone (android) to the server on PC - would that work too? And what need to be done from security point of view on server side? Do you have an example for these topics? I tried the example "Chat Server" shipped with LC Sample Stacks and got troubles to run when seperated into client and server apps. Hope this site is still alive, but maybe not, as Simon didn't got a reply for more than a year now.

Mark Wieder Sunday Apr 05 2020 at 06:37 PM

Giorgio, Simon - in the example above, the "onClientConnected pSocket" returns the host address in the form host:port, so the IP address can be extracted there.