Creating a Socket Connection in FlexSim

FlexSim can be the server or the client. In this example, FlexSim is the server and the outside application is the client.

The steps to communicate with FlexSim via sockets (and the corresponding FlexSim commands) are:

1. Start the Windows background processes – `socketinit()`
2. Create a socket – `servercreatemain(int port)`
3. Create a connection – `serveraccept(int noblocking)`
4. Accept messages – `serverreceive(int connection, char *buffer, int bufferSize, int noblocking)`
5. Close the connection created in step 4 – `servercloseconnection(int connection)`
6. Close the socket created in step 2 – `serverclosemain()`
7. Stop the Windows background process – `sockend()`

`socketinit()` – initializes a background process in Windows. This must be called before FlexSim can even create a socket connection to send or receive messages. This function returns a True if the initialization was successful.

`servercreatemain(int port)` – creates a socket that listens to the specified port. Some ports are frequently used by Windows. These "well-known" ports should be avoided. Typically, anything over 1024 is safe to use.

`serveraccept(int noblocking)` – FlexSim will attempt to accept a connection. If the noblocking parameter is a 0, FlexSim will freeze until a successful or failed response is received from the client. The purpose is to prevent FlexSim from doing anything if the client has not started yet.

This function returns an integer that serves as a unique identifier for this connection that should be stored for later use. If a connection is not successful, a 0 is returned.

`serverreceive(int connection, char *buffer, int bufferSize, int noblocking)` – receives a single messages from the specified connection. `Serverreceive()` acts differently in FlexSim and C++. In FlexSim, *buffer and bufferSize are not used. *buffer should be NULL and bufferSize can be any integer. The return value is a string of the actual message received.

In C++ *buffer is a char pointer where the message will be stored and bufferSize limits how big a message is stored. The return value is the number of bytes received.

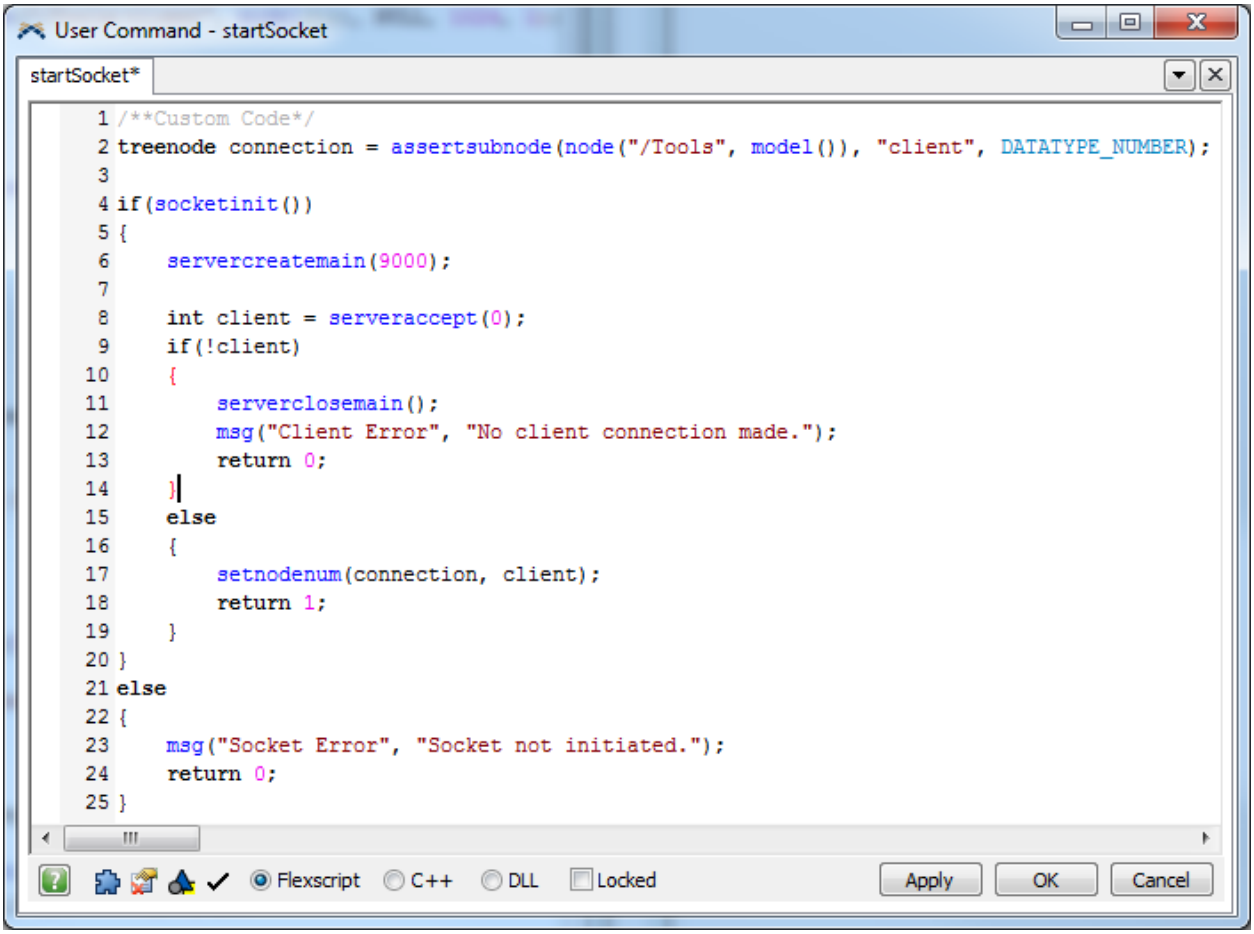
`servercloseconnection(int connection)` – closes the specified port.

`serverclosemain()` – closes the main server socket and all of the connections that are still open. No more communication can be done until `servercreatemain()` is called again. All connections should be closed individually (using `servercloseeconnection()`) before this is called.

Sample Model

Creating a Socket

In the sample model, in order to simplify socket creating and implement exception handling, `socketinit()`, `servercreatemain()`, and `serveraccept()` are in the startSocket user command.



```
treenode connection = assertsubnode(node("/Tools", model()), "client", DATATYPE_NUMBER);
//a global reference to store the connection identifier created by serveraccept()

if(socketinit())//socketinit() returns True if the background processes started
{
   servercreatemain(9000);//port 9000 is hardcoded, but could be a parameter

   int client = serveraccept(0);//needs to be blocking in case the client hasn't started
   //if a connection is made, serveraccept() will return a unique identifier
   //if a connection is not made, serveraccept() will return a 0

   if(!client)//the connection was unsuccessful
   {
      serverclosemain();//if not closed, servercreatemain() will throw an exception
      msg("Client Error", "No client connection made.");
      return 0;//return a 0 to indicate a socket connection was not created
```

```
    }
    else//if the connection was succesfully made
    {
       setnodenum(connection, client);//store the connection ID
       return 1;//return a 1 to indicate a socket connection was not created
    }
}
else//socketinit() was not successful
{
   msg("Socket Error", "Socket not initiated.");
   return 0;//return a 1 to indicate a socket connection was not created
}
```

Note: `serverclosemain()` is called if the connection was not successful. This is probably not be the best approach to exception handling because all open connections should be closed (using `servercloseconnection()`) before calling `serverclosemain()`. If an individual connection is not closed before calling `serverclosemain()`, weird things happened with sockets that could only be fixed by closing and reopening FlexSim.

We made the assumption that only one socket connection would be made in this model and since the connection was not successful, we could close the main connection here.

Listening to the Socket

After a socket is successfully created, FlexSim must listen for messages on that socket. The button Connect Client on the custom GUI calls the startSocket user command and then starts the timer:

```
startSocket();
createview("MAIN:/project/model/Tools/TimerObj");
```

TimerObj is actually a view and is located in the Tools node. When the view is created, the OnOpen node executes the following:

```
applicationcommand("settimer", c, 100);
```

which starts the timer and executes the OnTimerEvent in 100 miliseconds. The OnTimerEvent will continue to execute every specified number of milliseconds as defined in the time variable. The timer execute until the OnClose node executes the following:

```
applicationcommand("killtimer", c);
```

The actual listening is done by the OnTimeEvent using the following:

```
string message = serverreceive(getnodenum(node("/Tools/client", model())),
NULL, 1024, 1);
```

The variable message will contain the text received via the connection defined by the first parameter (`getnodenum(node("/Tools/client", model())`). Since this will be executed many times a second, most of the time message will be blank. So we must check to see when a message was received:

```
if(!comparetext(message,""))
```

```
{
   //do something with message
   serversend(getnodenum(node("/Tools/client", model())), "ask");
}
```

The last line is not necessary. It is just an acknowledgment sent back to the client so it knows the message was received. This needs to happen only if the client requires it.

After all messages have been received, it is important to close the socket connection(s). The sample model uses a user command called `endSocket()` to execute the following:

```
servercloseconnection(getnodenum(node("\Tools\client", model())));
serverclosemain();
socketend();
```

Notice the first parameter in servercloseconnection() is the reference to the connection identifier created and stored when the socket connection was created. In the sample model, `endSocket()` is executed in the OnClose of the TimerObj.