Networked Communication of Second Life® Objects using LSL, php, and MySQL.
By L. Christopher Bird (aka ZenMondo Wormser)

Part the First: The Problem and Solution

What do you do when you have a series of devices in the virtual world of Second Life each generating a small amount of data while performing a task that needs to be collected, acted upon and communicated to all other devices in real time?

In 2009, I approached the problem by making a hub and spoke network. This facilitated communication between Second Life objects and a central database where all the data was processed. before passing the result back out to the virtual objects via HTTP.

Part the Second: Background on Second Life scripting in LSL

If an object in Second Life does something, it is a script doing it. If it moves, talks, interacts, or do anything beside just sit there, it is scripted.

A script is basically a small computer program written in LSL (Linden Scripting Language), which defines an object's behavior.

To briefly explain the language we will look at a simple script “Hello Avatar”.

```
default
{
    state_entry()
    {
        llSay(0, "Hello, Avatar!");
    }

    touch_start(integer total_number)
    {
        llSay(0, "Touched.");
    }
}
```

The Hello Avatar script contains the three Elements necessary for all LSL scripts. These are a state, event handlers, and instructions.

States:

LSL is a state-based Language. All LSL scripts must have at least one state. In the Hello Avatar script we have One state, called "default". It is in red text. The code-block for the state is between the first opening brace "{" and the last closing brace "}".

The State contains the Event Handlers.

LSL is event-driven. When an event occurs, the code-block contained within the event handler is executed. The script Hello Avatar has Two event handlers. One named "state_entry" and one named

"touch_start". They are in blue text. The code block for each event handler begins with an opening brace, "{" and ends with a closing brace "}". The label for each event handler ends with a set of parenthesis "()". The parenthesis contain the elements that are passed to the event handler. In the state_entry event handler, there is nothing passed so the parenthesis are empty. In the touch_start event handler, it is passed an integer named "total_number" . So "integer total_number" is found within the parenthesis.

The Event Handlers contain the Instructions.

Each event handler in the Hello Avatar script contains one instruction. In this case the instruction is the function llSay. Instructions are the part of the script that we actually do things. All instructions end with a semi-colon ";" .

So the structures of an LSL script can be viewed like an onion. The outermost layer is the state, inside the state are found the event handlers, inside the event handlers are found the instructions.

The flow of Hello Avatar is as follows:

When the script is first run, or reset, the state default is entered. This triggers state_entry event. The state_entry event handler then executes the instructions in its code-block. In this case the the instruction llSay(0, "Hello, Avatar"); In the world, the object containing the script types "Hello, Avatar!" To Channel 0, which is the local chat channel.

When the object is touched (usually, clicked upon with the mouse cursor), this triggers the touch_start event. The touch_start event handler executes its code-block, in this case the instruction llSay(0, "Touched."); This happens every time the object containing the script is touched.

Part the Third: The Method and the Madness

Communication in the hub and spoke network is done over HTTP. The Second Life objects communicate to the central database by sending an HTTP request to a php page hosted on a server which then inserts the data into the database. The server then communicates to the network of Second Life objects by sending a HTTP request to the Second Life objects which use the LSL HTTP server (for more information see: http://wiki.secondlife.com/wiki/LSL_HTTP_server)

One of the shortcomings of the LSL HTTP server, is that URLs are not persistent. So one of the first things we must do is create a method of tracking object URLs when they are acquired or changed. The URL for each object in our network existing in the virtual world of Second Life is stored in an MySQL table. This is done by reporting this to a php script on our web-server which then inserts or updates the table.

Here is an example of a table structure used in the 2010 Relay For Life of Second Life Donation System:

```
-- --------------------------------------------------------

--
-- Table structure for table `kiosk`
--
```

```
CREATE TABLE IF NOT EXISTS `kiosk` (
  `kioskID` int(11) NOT NULL auto_increment,
  `UUID` varchar(36) NOT NULL,
  `region` varchar(128) NOT NULL,
  `position` varchar(64) NOT NULL,
  `teamnumber` int(11) NOT NULL,
  `kioskowner` varchar(128) NOT NULL,
  `kioskname` varchar(128) NOT NULL,
  `URL` varchar(512) NOT NULL,
  `active` tinyint(1) NOT NULL default '0' COMMENT 'Flag for Active
Kiosks/Vendors',
  PRIMARY KEY  (`kioskID`)
) ENGINE=MyISAM  DEFAULT CHARSET=latin1 AUTO_INCREMENT=5345 ;
```

The fields of interest are `UUID`, `URL`, and `active`. The UUID field contains a unique identifier for the Second Life object. The URL field contains the URL of the LSL HTTP server in the Second Life object and the active field contains a flag for if the Second Life object is currently communicating in the network. As a note of interest in this example the field `teamnumber` is used to divide the overall network into smaller sub-networks which contains a subset of the data aggregate.

A typical entry in the table may look like this:

```
(2708, '88b53b74-e01a-1862-bb2d-f77665ac4cda', 'Caledon On Sea',
'<28.48875,232.51993,39.10678>', 76, 'ZenMondo Wormser', 'RFL of SL
2010 Official 1-Prim Kiosk',
'http://sim5644.agni.lindenlab.com:12046/cap/0758de00-2d36-56e1-b619-
d152142bffa4', 0)
```

The communication of the URL to the database is handled by parts of a monolithic LSL script, I will reproduce the pertinent code segments here.

```
state registerKiosk
{
    state_entry()
    {
       llSetText("Registering...", <1.0, 1.0, 1.0>, 1.0);

       kioskOwner = llKey2Name(llGetOwner());

       llHTTPRequest("http://virtualrelay.org/registerthekiosk.php?
uuid=" + (string)llGetKey() + "&region=" +
llEscapeURL(llGetRegionName()) + "&position=" + (string)llGetPos() +
"&team=" + (string)teamNum + "&owner=" + llEscapeURL(kioskOwner) +
"&kioskname=" + llEscapeURL(llGetObjectName()) + "&pixie=bug_fairy",
[HTTP_METHOD,"GET"],"");

       if(myURL!= "")
       {
```

```
           llReleaseURL(myURL);
        }

        llRequestURL();

    }

    http_request(key id, string method, string body)
     {
        if (method == URL_REQUEST_GRANTED)
        {
            myURL= body;

            llHTTPRequest("http://virtualrelay.org/setURL.php?uuid="
+ (string) llGetKey() + "&url=" + myURL, [HTTP_METHOD,"GET"],"");


            state running;
        }

        else if (method == URL_REQUEST_DENIED)
        {
            llSay(0, "This Parcel has no available URLs and this
kiosk needs one to function properly. Please rez a kiosk on another
parcel or free up this parcel's available URLs.");
            llSleep(3.0);
            llDie();

        }

    }

    changed(integer change)
    {

        if(change & CHANGED_REGION_START)
        {
            llRequestURL();
        }

        if(change & CHANGED_OWNER)
        {
            llResetScript();
        }

    }

    on_rez(integer start_param)
    {
```

```
        llResetScript();
    }
}
```

(Similar code is found in other states to catch the request of a new URL in a changed event handler and the results handled in that state's http_request event handler.)

Here is the contents of the script setURL.php which handles the request generated by the Second Life object.

```
<?php require('header.inc'); ?>
<?php
$_GET{'uuid'} = mysql_real_escape_string($_GET{'uuid'});
$_GET{'url'} = mysql_real_escape_string($_GET{'url'});

$updatequery = "UPDATE `acsnho2_sl`.`kiosk` SET  URL = '" .
$_GET{'url'} . "',  active = '1' WHERE UUID = '" . $_GET{'uuid'} .
"'";

mysql_query($updatequery) or die ("Update Failed");



?>
<?php require('footer.inc'); ?>
```

(This script uses UPDATE because the field in the table is pre-populated with a place-holder URL in the registerthekiosk.php script.)

Pretty simple, right? Now with this information, we are ready to communicate with our network of Second Life Objects.

In the example application, the Relay For Life of Second Life 2010 Donation System. donations were collected by a variety of kiosks and vendors, routed to a single account in Second Life, and the details of the donation recorded in the database. Totals of the donations were done using MySQL queries and communicated to all kiosks and vendors for a specific team whenever a donation was made.

To communicate a donation to the server a single line of LSL was used in the money event handler.

```
llHTTPRequest("http://virtualrelay.org/realdonation.php?amount=" +
(string) amount + "&team=" + (string)teamNum + "&sponsor=" +
llEscapeURL(llKey2Name(id)) + "&owner=" + llEscapeURL(kioskOwner) +
"&region=" + llEscapeURL(llGetRegionName()) + "&position=" +
(string)llGetPos() + "&uuid=" + (string)llGetKey() + "&kioskname=" +
llEscapeURL(llGetObjectName()) + "&pixie=bug_fairy",
[HTTP_METHOD,"GET"],"");
```

The script realdonation.php then records the donation and communicates to the network using cURL. Here is the code snippet of the communication.

```
/* HTTP PUSH TO TEAM KIOSKS */

//First get team total
$totalquery = "SELECT sum(donationAmount) donationTotal FROM donation
WHERE teamnumber = '" . $_GET{'team'} . "'";

$totallist = mysql_query($totalquery) or die ("total query failed");

$total = mysql_result($totallist, 0);

$urlquery = "SELECT * FROM kiosk WHERE teamnumber = '" .
$_GET{'team'} . "' AND active = '1'";

$urllist = mysql_query($urlquery) or die ("url query failed");

while($url = mysql_fetch_array($urllist, MYSQL_ASSOC))
{
     $ch = curl_init();
     curl_setopt($ch, CURLOPT_URL,  $url['URL'] . "/?
msg=update&data=" . $total );
     curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1); //output to string
     $output = curl_exec($ch);
     curl_close($ch);

     if($output != "Hello World") //Kiosk did not repsond
     {
          $updatequery = "UPDATE `acsnho2_sl`.`kiosk` SET  active =
'0' WHERE URL = '" . $url['URL'] . "'";
          mysql_query($updatequery) or die ("update query failed");
     }

}
```

As you can see, the actual data is sent to the LSL HTTP server on the URL using the GET method. In this example application, 3 types of messages were accepted with the GET method. The above example sent the total update, the other messages being a verification phrase and the end of season message to remove the kiosk from world. These messages were handled in LSL by the http_request event handler which is excerpted below. (*You will recognize some of the method handling from an earlier excerpt.)*

```
http_request(key id, string method, string body)
     {
        if (method == URL_REQUEST_GRANTED)
        {
            myURL= body;

            llHTTPRequest("http://virtualrelay.org/setURLphp?uuid=" +
(string) llGetKey() + "&url=" + myURL, [HTTP_METHOD,"GET"],"");
```

```
        }

        else if (method == URL_REQUEST_DENIED)
        {
            llInstantMessage(llGetOwner(), "This Parcel has no
available URLs and this kiosk needs one to function properly. Please
rez a kiosk on another parcel or free up this parcel's available
URLs.");
            llSleep(3.0);
            llDie();

        }

        else if (method == "GET")
        {


            string query_string = llGetHTTPHeader(id, "x-query-
string");

            //llSay(0, "Query=" + query_string);

            list queries = llParseString2List(query_string, ["&"],
[]);

            //Break the list from the web into each element
            //This depends on the url talking to the kiosk to be
            //formed correctly

            list querytype =
llParseString2List(llList2String(queries, 0), ["="], []);
            list queryvalue =
llParseString2List(llList2String(queries, 1), ["="], []);

            string type = llList2String(querytype, 1);
            string value = llList2String(queryvalue, 1);

            //llSay(0, "Type=" + type + " Value=" + value);

            //Now Evaluate type and take action

            if(type == "update") //Team Total Update
            {
                teamTotal = (integer) value;
                kioskDisplay();
                llHTTPResponse(id, 200, "Hello World");
            }
```

```
            else if(type == "verify") //Kiosk Verification
            {

                llSay(0, "Your personal verification phrase is: " +
llUnescapeURL(value));
                llHTTPResponse(id, 200, "Your Verified Kiosk has said
your personal verification phrase in world.");

            }

            else if(type == "die") //End of Season Message
            {
                llHTTPResponse(id, 200, "Good Bye, Cruel World!");
                llDie();
            }

        }

    }
```

Part the Fourth: Conclusions and Musings

I developed this method in late 2009, and tested in early 2010, finally deploying it in the 2010 Relay For Life Donation system, which handled hundreds of thousands of transactions with very little server or network load. I believe this to be a fast and powerful method of communication between Second Life objects. I hope you have found this article informative, and the method useful.

About the Author:
L. Christopher Bird is the Hacker known as Johnny Fusion who drives the Avatar ZenMondo Wormser in the virtual world of Second Life and other grids. He wrote his first computer program on an Apple II in 1978 at the age of 6.