Towards a Distributed Simulation Toolbox for Scilab

Lin Wang [1], Umut Durak [1,2], Sven Hartmann [1]

[1] Clausthal University of Technology

Department of Informatics

[2] DLR Institute of Flight Systems

{wang.lin, sven.hartmann}@tu-clausthal.de, umut.durak@dlr.de

Abstract

In this document, I have made a further research on the toolbox about transferring data. The goal is to implement transferring a JSON(JavaScript Object Notation) from one terminal to another. In the first part, I make a introduction about Scilab. Then I will present the implements of my project.

1. Introduction

Scilab is free and open source software for numerical computation providing a powerful computing environment for engineering and scientific applications [1].
Scilab uses interpreted language to implement functions. This generally allows to get faster development processes, because the user directly accesses a high-level language, with a rich set of features provided by the library [2]. The Scilab interpreted language is meant to be extended so that user-defined data types can be defined with possibly overloaded operations. Scilab users can develop their own modules so that they can solve their particular problems. The Scilab language allows to dynamically compile and link other languages such as Fortran and C: this way, external libraries can be used as if they were a part of Scilab built-in features. Scilab also interfaces LabVIEW, a platform and development environment for a visual programming language from National Instruments [3].
Distributed simulation is a technology that enables a simulation to be executed on multiple computing nodes, such as a set of networked personal computers [4].

1.1 Toolbox in Scilab

Scilab is distributed with source code, so we can easily get to know the internal aspects of Scilab [5]. It offers an API to extend the language with C, C++ or Fortran code(sources or libraries). A toolbox have the following structures:

(1) Builder: users must first build it by launching the *builder.sce* script. Builder is to execute functions in other file(C files and Gateway files) and automatically generate a library and a loader. After execute *builder.sce*, as presented in Code 1, the main directory now contains a *loader.sce*.

```
files = ['udpserver.c','sci_udpserver.c','udpclient.c','sci_udpclient.c'];

functions = ['udpserver','sci_udpserver';'udpclient','sci_udpclient'];

ilib_build('network_build_c',functions,files,[]);

// load the shared library

exec loader.sce;
```

Code 1. Builder build the module

(2) Loader: after execute the *loader.sce*, user can now use the module,as if the functions were build in Scilab.

(3) Function file: developers can use the function file to simplify reuse functions.

(4)*readme.txt:* for this project, it described how to use the functions files.

(5)C files: logical files of the whole project. *udpclient.c* and *udpserver.c* are both written in C Language.

(6)Gateway files: the link between Scilab and application code is called Gateway. The gateway function is responsible of checking, converting, and transmitting data from Scilab to this external function (and reversely from the external function to Scilab), and calling this function[6]. *sci_udpclient.c* and *sci_udpserver.c* are the Gateway files in this project. They both written in C Language.

1.2 Preparations

To use Scilab, first you need to download it. My project runs only under linux system(the reason is still unknown). Therefore, you can either using linux system, or download a virtual box(with linux system).
The second step is to download Scilab. Here is the page to download latest version of Scilab:
http://www.scilab.org/download/latest

1.3 Description of JSON

JSON(JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language[7].
The JSON format is often used for serializing and transmitting structured data over a network connection. It is primarily used to transmit data between a server and web application, serving as an alternative to XML[8]. In Scilab, users can use *struct* to create a JSON.
For example:

json_struct = struct('Name','Lin','Password','123456')

Use this statement above, we can build a JSON:

{"Name":"Lin","Password":"123456"}

1.4 Task

Use struct to build a JSON, and transfer it from a client to a server. Using two terminals to simulate two ends: a client end and a server end.

1.5 Basement

My project is based on a existed transfer system: the former has already implemented to transfer double type and string type between two ends.

1.6 Problem & Solution

The problem is the console can not abstract a struct type date.
To solve this problem, I have got an idea:
in the client side, to change a struct type data into a string type, cause the string type can be transferred(the former has already made it); in the server side, receive the string type data,then convert them back to struct type and present.

2. implement

- **Fuctions in C files**

(1) Client side

There are 3 native function in *udpclient.c* file: *UDPsend_dblData for* sending double data; *UDPsend_file* for sending files; *UDPsend_strData* for sending text(string) data. Code 2 is an example of a native function, *UDPsend_strData*, written in pure C Language. It opens a UDP socket and sends the data with given address and port number, then close.

```
/* send text data using UDP */
int UDPsend_strData(char *_stServerName, int _iPort, char* _strData)
{
  int sock, length, n;
  struct sockaddr_in server;
  struct addrinfo hints,*res;
  char str[INET_ADDRSTRLEN];

  memset(&hints,0,sizeof hints);
  hints.ai_family=AF_INET;
  hints.ai_socktype=SOCK_DGRAM;

  /*double buffer[1];*/
  if((n=getaddrinfo(_stServerName,NULL,&hints,&res))!=0)
      fprintf(stderr, "getaddrinfo error: %s\n",gai_strerror(n));

  sock = socket(AF_INET,SOCK_DGRAM,0);
```

```
  if (sock < 0)
        perror("socket failed!");

  server.sin_addr=((struct sockaddr_in*)res->ai_addr)->sin_addr;
  server.sin_family = AF_INET;
  server.sin_port = htons(_iPort);
  length=sizeof(struct sockaddr_in);

  /*send data using UDP*/
  n=sendto(sock, _strData, MAX_STR, 0, (const struct sockaddr*)&server, length);
  if (n < 0){
        perror("send failed!");
        printf("%s:%d",inet_ntop(AF_INET,&server.sin_addr,str,sizeof(str)),server.sin_port);
}
  else
        printf("UDP sent data= %s, n=%d\n", _strData, n);

   shutdown(sock,SHUT_RDWR);
}
```

Code 2. Native C function to send test data using UDP

(2) Server side

There are 2 native function in *udpserver.c* file: *UDPrcv_dblData*, UDPrcv_strData. *UDPrcv_dblData for* receiving double data; *UDPrcv_strData* for receiving test data. Code 3 is an example of a native function, *UDPrcv_strData*, written in pure C Language. It opens a UDP socket and receives the data with given port number, then close. The port number here must the same as it in the client side.

```
/* Start UDP Server function */
int startServer_UDP(int _iPort)
{
   struct sockaddr_in server,from;
   int sock = 0;
   int iFromLength = 0;
   int iServerLength = 0;

   void *pdblData;

   sock = socket(AF_INET, SOCK_DGRAM, 0);
   if (sock < 0)
        perror("socket failed!");
   else
         printf("socket created\n");

   server.sin_family=AF_INET;
   server.sin_addr.s_addr=INADDR_ANY;
   server.sin_port=htons(_iPort);
   iServerLength = sizeof(server);
   if (bind(sock, (struct sockaddr *)&server, iServerLength) < 0)
         perror("bind failed!");
   else
        printf("socket bound\n");
```

```
    return sock;
}

/* receive text data using UDP */
char* UDPrcv_strData(int _iSocket)
{
    struct sockaddr_in from;
    int iFromLength = sizeof(struct sockaddr_in);
    char* strData=(char*)malloc(MAX_STR * sizeof(char));
    int iByteReceive = recvfrom(_iSocket, strData, MAX_STR, 0,
   (struct sockaddr *)&from, & FromLength);
    if (iByteReceive < 0)
          perror("UDP-receive failed!");
    else
          printf("UDP-received data: %s       iByteReceive=%d\n",strData, iByteReceive);
         close(_iSocket);
    return strData;
}
```

Code 3. Native C function to send test data using UDP

- **Functions in Gateway**

Generally to build a Gateway file, here lists the necessary steps:
(1) check the number of arguments(input and output);
(2) manage input arguments, define the addresses and give them values to the certain variables:
a. get the variable address
b. check types of the variables
c. others are optional, such as other checks or data transformation
(3) add Application code
Below presents a Gateway file in Code 4, called sci_udpclient.c, showing how Gateway file created.

```
int sci_udpclient(char *fname)
{
        SciErr sciErr;
        int* portAddr = NULL;
        int* dataAddr = NULL;
        int* typeAddr = NULL;
        int* domainAddr = NULL;
        double port = 0;
        char *domain = NULL;
        double dblData = 0;
        char* strData = NULL;
        double type = 0;

CheckInputArgument(pvApiCtx, 4, 4);
CheckOutputArgument(pvApiCtx, 0, 1);

        sciErr = getVarAddressFromPosition(pvApiCtx, 1, &domainAddr);
        if(sciErr.iErr)
        {
                printError(&sciErr, 0);
```

```
        return 0;
    }
    sciErr = getVarAddressFromPosition(pvApiCtx, 2, &portAddr);
    if(sciErr.iErr)
    {
        printError(&sciErr, 0);
        return 0;
    }
    sciErr = getVarAddressFromPosition(pvApiCtx, 3, &typeAddr);
    if(sciErr.iErr)
    {
        printError(&sciErr, 0);
        return 0;
    }
    sciErr = getVarAddressFromPosition(pvApiCtx, 4, &dataAddr);
    if(sciErr.iErr)
    {
        printError(&sciErr, 0);
        return 0;
    }

    /* ====== check inputs ======= */
    // check domain
    if(!isStringType(pvApiCtx, domainAddr))
    {
        Scierror(999, _("%s: Wrong type for input argument #%d: A string expected.
\n"), fname, 1);
    return 0;
    }

    if ( getAllocatedSingleString(pvApiCtx, domainAddr, &domain) )
    {
        Scierror(999, _("%s: Wrong size for input argument #%d: A scalar expected.
\n"), fname, 1);
        return 0;
    }

    // check port
    if(!isDoubleType(pvApiCtx, portAddr))
    {
        Scierror(999, _("%s: Wrong type for input argument #%d: A Integer
expected.\n"), fname, 2);
    return 0;
    }

    if ( getScalarDouble(pvApiCtx, portAddr, &port) )
    {
        Scierror(999, _("%s: Wrong size for input argument #%d: A scalar expected.
\n"), fname, 2);
        return 0;
    }


    // check type
    if(!isDoubleType(pvApiCtx, typeAddr))
    {
```

```
                Scierror(999, _("%s: Wrong type for input argument #%d: A Integer
expected.\n"), fname, 2);
        return 0;
        }

        if ( getScalarDouble(pvApiCtx, typeAddr, &type) )
        {
                Scierror(999, _("%s: Wrong size for input argument #%d: A scalar expected.
\n"), fname, 2);
                return 0;
        }
```

Code 4. First 2 step for creating a Gateway file

And now using flag to identify different types of the data(1= double, 2= string). Destcription shows in Code 5.

```
        // check data
        int flag=0;
        if(isDoubleType(pvApiCtx, dataAddr))
            flag=1;
        else if(isStringType(pvApiCtx, dataAddr))
            flag=2;
        else
        {
                Scierror(999, _("%s: Wrong type for input argument #%d: A double/string
expected.\n"), fname, 3);
                return 0;
        }
```

Code 5. Distinguish different data type

Use given type(given by function files) to call functions. In Code 6, it calls *UDPsend_strData* in C file.

```
        // send text data
        else if(type==3)
        {       if ( getAllocatedSingleString(pvApiCtx, dataAddr, &strData) )
                {
                        Scierror(999, _("%s: Wrong size for input argument #%d: A scalar
expected.\n"), fname, 3);
                        return 0;
                }
                UDPsend_strData(domain,(int)port,strData);
                sciprint("Data sent (UDP): %s      size: %d\n",strData,sizeof(&strData));
        }

        return 0;
}
```

Code 6. Connect C Function

- **Functions in Scilab**

Code 7 and Code 8, *sendUDP.sci* and *receiveUDP.sci*, are the function files. *sendUDP* function is to include the *udpclient* function and make the JSON file easy created by users. Function file is started from "*function*" and ended with "*endfunction*". When user runs this file, the statements inside would be carried out.

```
function sendUDP(host_ip,host_port,json_struct)
        temp = getfield(1,json_struct);
        n = size(temp,'c');
        send_string = '{';
        for i=3:n
                send_string=send_string+ascii(034)+temp(i)
                +ascii(034)+':'+ascii(034)+getfield(i,json_struct)+ascii(034);
                if i < n
                        send_string = send_string + ','
                end
        end
        send_string = send_string + '}'
        udpclient(host_ip,host_port,3,send_string)
endfunction
```

Code 7. sendUDP function(use in client side)

```
function received_string = receiveUDP(host_port)
        received_string = udpserver(host_port,2);
endfunction
```

Code 8. *receive*UDP function(use in server side)

3. Result

The result windows in console are showed in the following figures, both after execute the *loader.sce*.

```
-->exec('/home/lin/下载/lastversion/receiveUDP.sci',-1)

-->receiveUDP(2000)

UDP Socket created and bound
Waiting for Data...
Data reveived (UDP): {"Name":"Lin","Password":"123456"}       size: 8!!
 ans  =

{"Name":"Lin","Password":"123456"}

-->
```

Figure 1. Server side

Here give the port number 2000, and wait for data. After get the data, display on the console.

```
-->json_struct=struct('Name','Lin','Password','123456')
 json_struct  =

   Name: "Lin"
   Password: "123456"

-->exec('/home/lin/下载/lastversion/sendUDP.sci',-1)

-->sendUDP("localhost",2000)
Data sent (UDP): {"Name":"Lin","Password":"123456"}      size: 8
```

Figure 2. Client side

Here to create a JSON, and runs function sendUDP using port 2000, the same as in server side.

```
-->f=receiveUDP(2000)

UDP Socket created and bound
Waiting for Data...
Data reveived (UDP): {"Name":"Lin","Password":"123456"}       size: 8!!
 f  =

{"Name":"Lin","Password":"123456"}

-->f
 f  =

{"Name":"Lin","Password":"123456"}

-->
```

Figure 3. Reuse data in server side

This is just a process about giving value to *f*, then execute *f* to redisplay the JSON.

4. Conclusion

This paper present a peer-to-peer network capability for Scilab by building a distributed simulation toolbox in Scilab.
In this project I successfully build a JSON, refactor the inputs in consoles by using function files , connect Scilab console's inputs to application code by using Gataway files, and transfer data from client to server by using sockets in c files.
Further working is to develop a full capability implementation for scilab communication.

5. References

[1] Scilab help. [Online] Available at: http://www.scilab.org/scilab/about [Accessed on 18 August, 2016]

[2] C. Bunks, J-P. Chancelier, F. Delebecque, M. Goursat, R. Nikoukhah and S. Steer. Engineering and scientific computing with Scilab. Edited by Claude Gomez. Springer Science & Business Media, 2012.
[3] Scilab introscilab. [Online] Available at:http://www.scilab.org/content/download/247/1702/file/introscilab.pdf [Accessed on 26 August, 2016]
[4] R. M. Fujimoto. Parallel and Distributed Sim- ulation Systems. Proceedings of 2001 Winter Simulation Conference, Arlington, VA, 2001.
[5] S. L. Campbell, J. P. Chancelier and R. Nikoukhah. Modeling and Simulation in Scilab/Scicos. Springer, USA, 2006.
[6] Scilab gataway. [Online] Available at: https://wiki.scilab.org/howto/Create%20a%20gateway [Accessed on 26 August, 2016]
[7] Ecma International,ECMAScript® 2016 Language Specification, [Online] Available at: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf[Accessed on 26 August, 2016]
[8] Json. [Online] Available at:https://atoms.scilab.org/toolboxes/json [Accessed on 26 August, 2016]