libm3l and lsipdx - Utilities for Inter-Process Data Transfer and Synchronization

Adam Jirásek[*], Arthur Rizzi[†]

The Royal Institute of Technology KTH, Stockholm, Sweden

This paper describes two Open Source Software libraries which are currently under development - *libm3l* and *lsipdx*. The *libm3l* library is a utility enabling storage of the basic data types in a linked list and their transfer through the TCP/IP socket. The linked list can be transferred as a whole or just its part. The second utility, the *lsipdx* library, is a utility which enables transfer of the data among a number of processes and their synchronization. The libraries are therefore suitable candidate for data exchange and synchronization of a number of solvers in the multidisciplinary analysis. Both libraries are written in the ANSI-C programming language and are available to the public under GNU Lesser GPL, LGPL, license through the github site.

Nomenclature

lsipdx	inter-process data exchange utility
libm3l	multi-level linked list library
TCP/IP	transmission control protocol / Internet protocol
OSS	Open Source Software
$LGPL$	Lesser LGPL license

I. Introduction

The increasing demand of multidisciplinary analysis in the aerospace research and industry puts a special demands on existing and new solvers and the coupling software which is used to enable communication and synchronization of the individual components. In recent years, the development effort on common communicating software platforms has been rising and some of the projects are rather large such as the CREATE,[1,2] GMSEC[3,4] or SALOME.[5,6]

Currently, most solvers used in certain area of the aeronautical engineering have been usually developed by specialists in this area rather then in computer science and solvers are usually stand-alone applications. These solvers can be parallelized and can be run efficiently on a large number of processors. However mature, most of them lack one capability - an efficient communication with another solver.

The coupling can be done in two ways - monolithic and partitioned coupling. The monolithic coupling requires the two problem to be formulated an implemented in a monolithic code. Such a code requires a substantial amount of time to be implemented, the line count of the code is very large and the code is difficult to maintain. Furthermore, any additional feature which is to be implemented requires substantial changes of the code. The biggest dissadvantage of this approach is, however, a lack of modularity and low flexibility.

The other approach, a partition coupling works with separate processe which share necessary information. This approach can make use of stand alone codes which are then coupled during the run. Each code is usually dedicated to solving one area of problem and uses solution of other codes as an input. It then produces thwe output which is used as an input to other processes. This approach is easier then the first one, however the hidden complication is in multiple solver communication and synchronization.

[*]Affiliated Researcher, Department of Aeronautical & Vehicle Engineering, AIAA Senior Member

[†]Professor, Department of Aeronautical & Vehicle Engineering, AIAA Associate Fellow

One way, possibly the most intuitive is a communication through file I/O operations, ie. every time the solvers need to communicate they save and read files with the required data set. All communicating solvers have to have an access to a common physical device. In addition, using disk I/O device can be rather slow in particular for larger data sets. The disk I/O operation itself does not guarantee synchronization of processes accessing a common file, the synchronization has to be done through the file locking operations.

Another approach is to use the Message Passing Interface, MPI.[7] Such an approach is very efficient, however requires time to be implemented and its generalization is rather difficult. Moreover the implementation is not easy and is prone to errors[8]

Recently, a Python[9] based approach for inter-process communications has been successfully tested. The python is a higher level interpreter language and in coupling problems is used as a wrapper for all processes which are communicating. The communication itself is done through sockets.[10,11] This approach is today the most popular approach for the inter-process communication and synchronization in aeronautical engineering as shown in a number of its applications.[8,12,13,14,1,15,16]

This paper presents a contribution to the multi-solver communication and synchronization - the two newly developed OSS libraries - the *libm3l* and *lsipdx*.[17] The first library, *libm3l*, is a C-library with a specifically designed communication protocol for the data transfer through the TCP/IP socket. The data are stored in the linked list and the library enables doing operations with the data in the linked list such as adding, removing, renaming, linking etc. The second library, *lsipdx*, enables efficient communication among a number of processes and their synchronization. The number of data sets which are shared as well as a number of communicating processes is not limited. The blocking sockets which are used for the data transfer guarantees the synchronization of the communicating processes. The emphasis is put on easiness of implementation and minimal required changes into existing codes which are being coupled and a possibility of a flexible and modular, multi-layer synchronization approach to the problem. The libraries are developed for the Linux/Unix type of operating systems and are complying with POSIX[18,19,20] standard.

II. Multi-level linked list TCP/IP transfer protocol - *libm3l*

The *libm3l* library is a library which enables creating and maintaining the data in a linked list. In addition the library enables transfer of the linked list between processes through the TCP/IP socket.

The basic element of the *libm3l* linked list is the *(node_t *)* node with a specific type and given name. The nodes are denoted according to basic type (for ex. F for float, D for double, I for integer, C for character), DIR if the node containing another branch of linked list or $LINK$ for the node which is a link to another node. The nodes store data in the one-dimensional or multi-dimensional arrays.

Figure 1 shows an example of a linked list. Each circle represents a node. The list in figure contains two

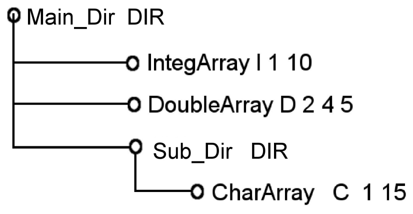


Figure 1. Example of *libm3l* linked list

DIR nodes called $Main\_dir$ and Sub_dir, the first being a head node (ie. the node at the very beginning of the list). The $Main\_dir$ node contains two arrays. The first is called *IntegArray* which is an array of one dimensional array of 10 integers. The second array is two dimensional array of double numbers with dimensions 4 and 5. In addition, the $Main\_dir$ DIR node contains another Sub_dir node type of DIR which contains the one dimensional char array of 14 characters plus one for an ending character of a char array $' \setminus 0'$.

The nodes in the linked list are then located using function *m3l_Locate()*. Following snippet of the code searches for the *Double_Array* node which is located under the $Main\_Dir/$. On return the function returns a pointer to structure *find_t ** identifies the type of the node, number of array dimensions, total number of array and gives a pointer to the array

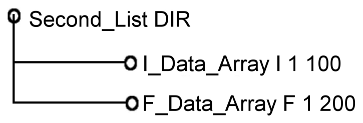
```
/*
 * look for Double_Array in Main/Dir
 * as an additional location information, the */* is specified which means
 * look for all possible /Main_Dir/Double_Array items in the list
 * if it happens that the list contains more /Main_Dir or
 * /Main_Dir/Double_Array then just one, the function locates
 * all of them
 */
if( (DATA_SFounds = m3l_Locate( (node_t *)Node, "/Main_Dir/Double_Array", "/*/*", (lmchar_t *)NULL)) != NULL){
/*
 * find how many nodes of the name Double_Array is in /Main_Dir, in this example it should be 1
 */
n_finds = m3l_get_Found_number(DATA_SFounds);
/*
 * get the pointer on the first found Double_Array and find details about the array
 */
FoundNode = m3l_get_Found_node(DATA_SFounds, 0);
/*
 * get details about the array
 */
type = lmchar_t *m3l_get_List_type(FoundNode);  /* get type of node, in this example should be 'D' */
ndim = m3l_get_List_ndim(FoundNode);  /* get number of array dimensions, in this example should be 1 */
array_dim = m3l_get_List_dim(FoundNode); /* get array dimensions, should be an int array with values
array_dim[0]=4 and array_dim[1]=5 */
nmax = m3l_get_List_totdim(FoundNode); /*  get total dimensions of array, in this example should be 20 (4X5) */
/*
 * get the pointer on the values of the Double_Array
 */
values = (double *)m3l_get_data_pointer(FoundNode)
/*
 * find the value of i,j-th array member, would an array be one dimensional, the value is obtained
*  accessing array values[i] trough direct indexing
 */
a = values[ m3l_get_2ind(i, j, array_dim[0], array_dim[1])  ];
}
else
{
printf("Server: did not find any requested Data_set\n");
exit(0);
}


m3l_DestroyFound(&DATA_SFounds);  /* free the structure which was allocated upon return from m3l_Locate()  */
```

The library enables operations with independent linked lists. The linked list than can be merged by moving one list to another, duplicated by copying a list to another list or linked with each other. As an example, Fig 2 shows a new linked list in Fig 2(a) which is linked the first linked list from Fig 1 into the *DIR Sub_dir* and an internal node linked to a different node in a different position in the same linked list The final list is shown then in Fig 2(b). The list *Second_list* is then accessible through its own head node or as a link through the $Main\_Dir/Sub\_Dir/Linked_List$ pointer. The *Char_Array* is then accessible through $Main\_Dir/Sub\_Dir/Char\_Array$ or through $Main\_Dir/Sub\_Dir/Link\_To\_InternalArray$ pointer.

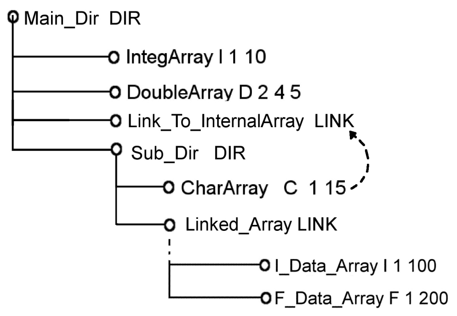
The *libm3l* linked list can contain nodes of the same name and type. An example is shown in figure Fig 3 The list contains several items called *Sub_Dir* which are of the same type *DIR* and each contains several data sets *Char_Array*. The locator function *m3l_Locate()* needs additional information such as the list content or the position in the linked list tree.

A. Transfer of the linked list to a separate process

The *libm3l* transfers the data to the separate processes through the TCP/IP Berkeley Unix Sockets[10,11] which are routinely used for communication over the Internet. The transfer is done using *libm3l* functions



(a) **List to be linked**



(b) *Main_Dir* **linked list with** *Second_list* **linked to it**

Figure 2. Linked list from previous example with link to another list

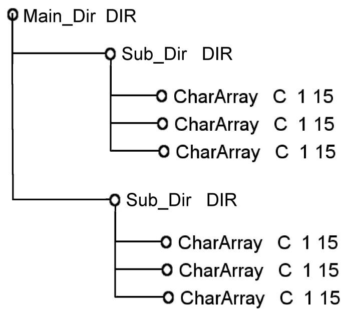


Figure 3. Example of *libm3l* linked list with nodes of the same type and name

which use as an input parameter a pointer to the node which is will be sent to socket. This node has to be always a *DIR* type of node. The second process receiving the data stored in the *libm3l* linked list then automatically create the identical copy of the list which was sent over the socket.

The sending of the data is done using function

```
Send_to_tcpipsocket( (node_t *)SendNode, Hostname, portnumber, Options ...)
```

with $(node\_t*)SendNode$ as a parameter. The receiving process then uses function

```
(node_t*)RecNode = Receive_tcpipsocket(Hostname, portnumber, Options ...)
```

Figure 4 shows a principle of the TCP/IP transfer of the data via *libm3l* library. The list which is to be transferred is the one from Fig 1. If the pointer *SendNode* points on the the node $Main\_dir$ then the receiving process receives an identical copy of entire list as shown in Fig 4(a). If the pointer *SendNode* points at node *Sub_Dir* then the receiving process receives a subset of the list the *Sub_Dir* as shown in Fig 4(b).

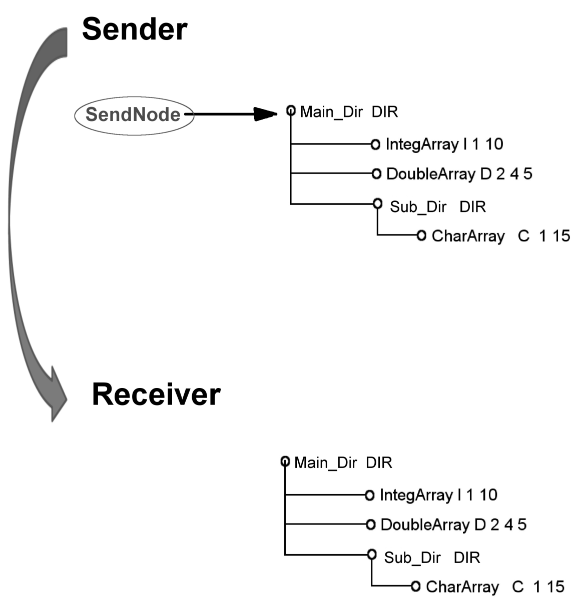
B. Invoking *libm3l* functions

Each function in the *libm3l* library can be called using an API which takes human readable parameters and uses *get_longopt* function to pars them or directly invoking a function with options specified in a structure *opt_t **. As an example, the function which sends the data to the TCP/IP socket can be invoked through API as

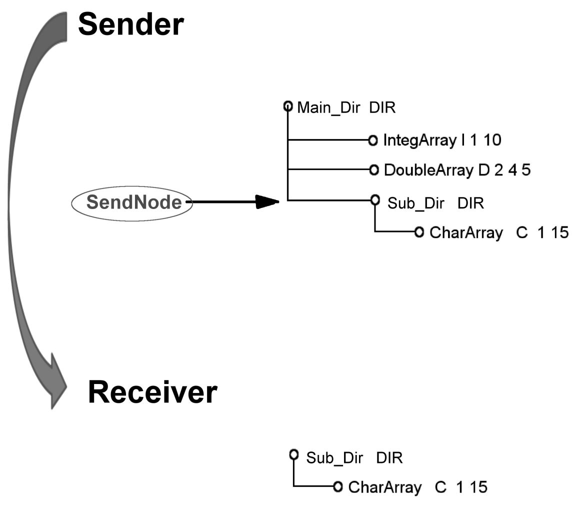
```
m3l_Send_to_tcpipsocket((node_t *)Node, hostname, sockfd, "--encoding" , "IEEE-754", (char *)NULL);
```

or by filling a structure with options and invoking directly the send function

```
m3l_set_send_to_tcpipsocket(&Opts);  /* set default parameters */
Opts->opt_tcpencoding = 'I';         /* set option --encoding IEEE-754  */
m3l_send_to_tcpipsocket((node_t *)Node, hostname, sockfd, (opts_t *)Opts);
```



(a) **Sending entire list**



(b) **Sending subset of the list**

Figure 4. Example of list sent over TCP/IP socket

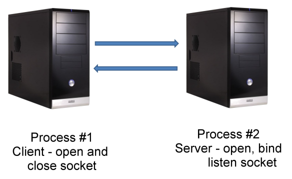
III. Synchronizing and Inter-Process Data Exchange Utility - *lsipdx*

In the previous section with an example of the *libm3l* data structure transferred through TCP/IP socket, the communication occurred between two processes. One process therefore resumed a role of a client and the second resumed a role of a server - see Fig 5(a). The two processes were not equal in the process hierarchy, one of the processes took care of opening, binding and listening a TCP/IP socket and then waiting until the second process opens a socket and start transferring the data, this is a simple Client-Server scenario. Such an arrangement is easy to implement however has several limitations, the biggest being that including more then two processes is rather difficult, in particular if the process which resumes a server role is primarily not a server dedicated software. A more suitable arrangement is shown in Fig 5(b) where the different processes are engaged in a communication with each other through a dedicated utility which takes care of data transfer and synchronization.

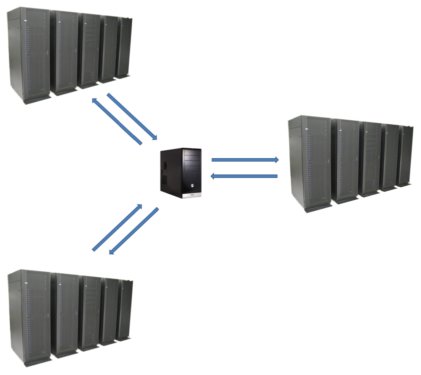
The *lsipdx* library is a utility taking role of a server which takes care of the data exchange between a sending process and the receiving processes the process synchronization. It opens required number of connections with a specific name and creates required number of hubs for each connection, one for the Sender and others for Receivers. During data transfer, each transferred data set is then uniquely defined by the name of the communication channel and each client process defines its role in the transmission chain by specifying if it is a sender or receiver. An arbitrary number of different data sets can be transferred and the transfer of one data set is independent of transfer of other data sets. The only dependency exists between processes associated to the same channel.

Figure 6 shows an example of a process which is intended to transfer three communcation channels denoted as *pressure*, *.density* and *temperature*. The the communication channel transfers data between the Sender and three receiving processes, the second channel transfers data between a sender and four receiving processes and the third channel between a sender and one receiving process.

Each circle in the figure shows a group of threads which takes care of the data transfer through the associated channel. This group is then taking care of data transfer between Sender and Receivers. Each arriving process establishes TCP/IP connection with the server thread and identifies itself with a name of the communication channel it will work with and its own mode - S for Sender and R for Receiver. The connection is then transferred to a given group of threads which waits until all processes working with the same data site arrive. As soon as this happens, the server starts data transfer. At the end of the transfer.



(a) **Simple server-client scenario**



(b) **Multiple process communication through one server**

Figure 5. Process communication

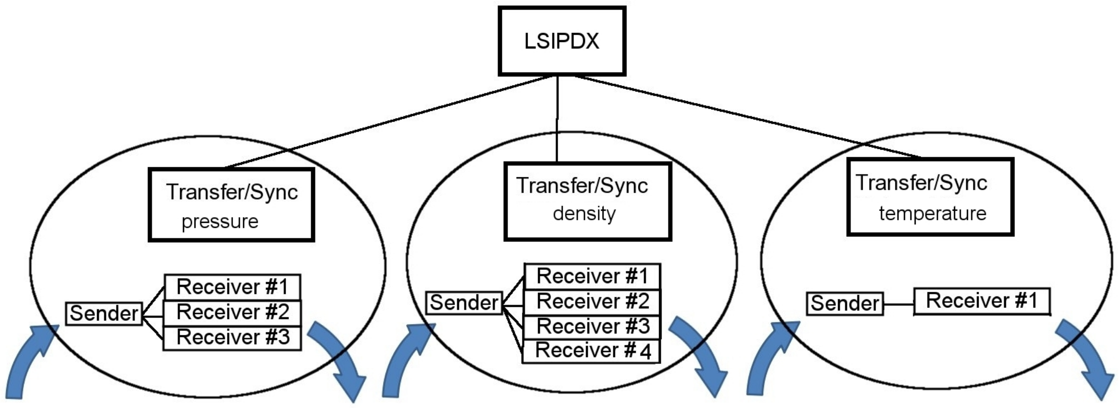


Figure 6. Schematics of *lsipdx* server data exchange

each process notifies the server that the data transfer was done correctly and proceed with another action. In such a way, the server controls weather all data were delivered correctly from sending process to all receiving processes and at the same time synchronizes the sender and Receivers.

A. *lsipdx* Server Initialization

The initialization file for the example from above is shown in following section.

```
COMM_DEF  DIR 2
        Process_Specifications DIR 1
                Name_of_Case  C 1 10
                        ‘Test Case‘
        Data_Sets  DIR 3
                Data_Set  DIR 4
                        Name_of_Data_Set C 1 9
                                ‘pressure‘
                        Sending_Process I 1 1
                                1
                        Receiving_Processes ST 1 1
                                3
                        CONNECTION DIR 2
                                ATDT_Mode C 1 2
                                ‘D‘
                                KEEP_CONN_ALIVE_Mode C 1 2
                                ‘N‘
                Data_Set  DIR 4
                        Name_of_Data_Set C 1 8
                                ‘density‘
                        Sending_Process I 1 1
                                1
                        Receiving_Processes ST 1 1
                                4
                        CONNECTION DIR 2
                                ATDT_Mode C 1 2
                                ‘D‘
                                KEEP_CONN_ALIVE_Mode C 1 2
                                ‘N‘
                Data_Set  DIR 4
                        Name_of_Data_Set C 1 12
                                ‘temperature‘
                        Sending_Process I 1 1
                                1
                        Receiving_Processes ST 1 1
                                1
                        CONNECTION DIR 2
                                ATDT_Mode C 1 2
                                ‘D‘
                                KEEP_CONN_ALIVE_Mode C 1 2
                                ‘N‘
```

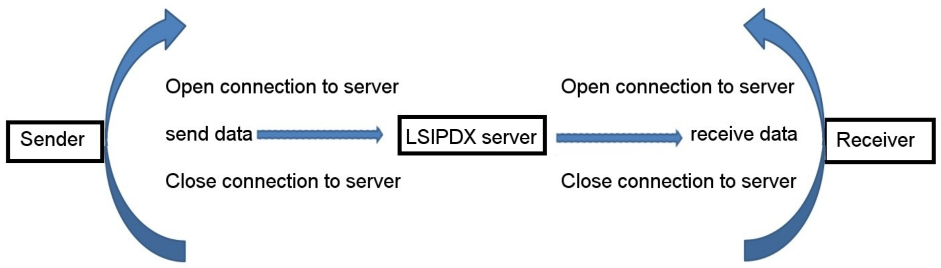
The server is asked to open three independent channels for each specified data set, and wait until a sending process and a number of receiving processes arrive. In addition is specifies two modes for each connections - the *ATDT* mode and *Keep_Connection_Alive* mode.

B. Modes of connections

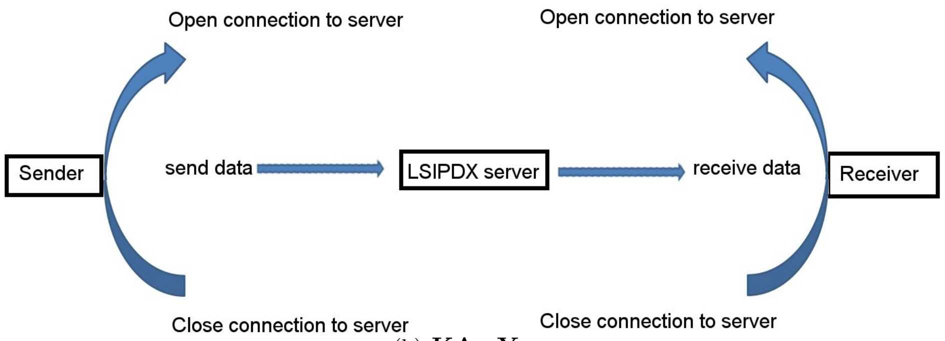
KA or *Keep_Connection_Alive* is a characteristics of the connection specifying how long the connection between a server and a client will exist. For *KA*=N the connection between a client and a server is opened when the data transfer is required and after the data transfer is finished the connection is closed for all processes taking part in the transfer. If a new transfer of the data set with the same name is required, the connection is opened and closed again.

For *KA*=Y the connection is opened at the beginning of the clients execution and is kept opened until the *lsipdx* server is terminated. Once the clients connect to this type of connection they have to keep it opened and can not reopen it.

Fig 7 shows schematically the *KA* mode function. The first mode is somewhat simpler to implement as it does not require passing the value of the descriptor and has an advantage of keeping the number of opened sockets at the necessary minimum as the sockets are used only during data transfer. The second mode is preferred in situations where there there is a high frequency of opening-closing socket in a loop.


Open connection to server
send data
Close connection to server
Sender
LSIPDX server
Open connection to server
receive data
Close connection to server
Receiver

(a) **KA=N**


Open connection to server
Open connection to server
Sender
send data
LSIPDX server
receive data
Receiver
Close connection to server
Close connection to server

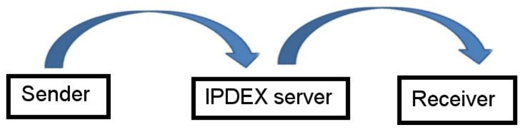
(b) **KA=Y**

Figure 7. Schematics of the different *KA* modes

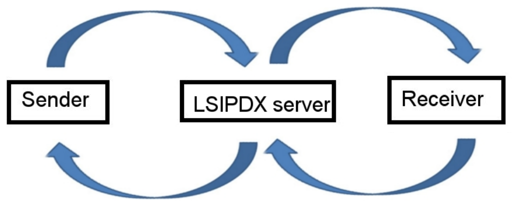
ATDT or *Alternate_Transfer_Direct_Transfer* is a characteristics specifying the direction of data flow through the connection to the server. The *ATDT*=D specifies the one directional flow, ie. a Sender client sends data through the server to Receiver client(s). Once the data is sent, both sending and receiving processes are notified about successful transfer and the transfer is considered as successfully finished.

The *ATDT*=A specifies the data flow is an alternate type of flow, ie. once a Sender client sends the data to a Receiver client, the two processes are notified the transfer was successful. Then the Sender and Receiver swap their roles and the Receiving process sends data set to the Sender client, once this reverse direction transfer is finished the entire data transfer is considered successfully finished.

Figure 8 shows schematically the *ATDT* modes. The limitation of the *ATDT*=A mode is that the connection exists between one Sender and one Receiver only.



(a) **ATDT=D**



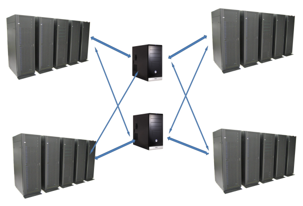
(b) **ATDT=A**

Figure 8. Schematics of the different *ATDT* modes

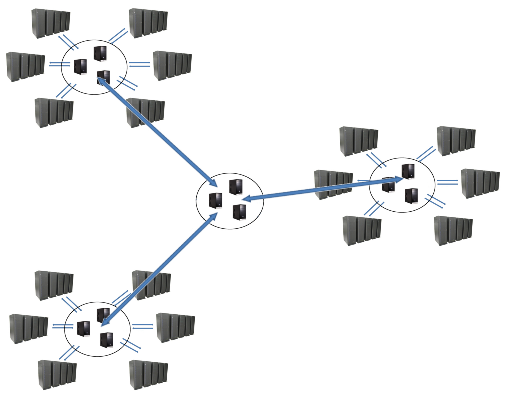
C. Multi-connection synchroinization

The examples above used an application scheme where all clients (Senders and Receivers) sharing data sets connect to one server which directs transfer of the data flow. Sometimes, it is however useful to segment problem of communication processes to groups which can greatly simplify the setup of entire communication. Such an arrangement is easily possible to do with the *lsipdx* utility.

Fig 9 shows the two possible communicating setup - the first showing several clients communicating through two synchronizing utilities, the second figure, Fig 9(b) shows multi-level communication of a number of processes.



(a) **Single transfer layer scheme**



(b) **Multiple transfer layer scheme**

Figure 9. Process communication with several synchronizing utilities

IV. Conclusion

This paper presents two Open Source Software, OSS, project currently under development which can be effectively used in the multidisciplinary modeling. It is available to potential users and developers on github web page in the *libm3l* and *lsipdx* repositories[21] and is released under the GNU LGPL license. The libraries are not tied to any specific application and can be used for any type of application requiring sharing data between separately run processes. Both libraries are easy to implement and require minimal changes in the source code of communicating processes.

The first library is the *libm3l* library which works with data in a linked list and enables their transfer through the TCP/IP socket. The data sets are stored as individual nodes in the linked list. The list in a linked list can be moved, copied, removed, added and linked to another linked list or to a different position in the same list.

The second library is a lsipdx which enables the data transfer and synchronization of the data sets among a number of processes. The data are transferred between the connecting processes through TCP/IP connection and each data set transfer is independent of transfer of other data sets. The connection can be characterized by two modes determining how long the connection exists and the data flow direction. The proposed communicating utility enables the modular approach to the design of the communication and synchronization scheme, ie. it enables using independent synchronization point to control the overall data flow. Such a feature has consequences on the design of data flow as well as to be used for designing the redundancy in a data flow and synchronization scheme.

The libraries are using ANSI C programming language and were tested on Linux and Unix operating system. In one application the *libm3l* library was used to transfer data to an application running on Windows system, however the update for the Windows environment were temporary fix and are not part of the official distribution.

References

[1]D. E. Post and S. Arevalo and C. Atwood and P. Bell, and T. D. Blacker and S. Dey and D. Fisher and D. A. Fisher and P. Genalis and J. Gorski and A. Harris and K. Hill and M. Hurwitz and R. P. Kendall and R. L. Meakin and S. A. Morton and E. T. Moyer and R. Strawn and D. van Veldhuizen and L. G. Votta and S. Wynn and G. Zelinski, “A new DoD initiative: the Computational Research and Engineering Acquisition Tools and Environments (CREATE) Program,” Journal of Physics: Conference Series 125(1).

[2]“DoD HPCMP Computational Research and Acquisition Tools and Environment (CREATE) program,” DoD HPC Insights, Spring 2013.

[3]“GMSEC - Goddard Mission Service Evolution Center,” http://gmsec.gsfc.nasa.gov/, accessed dec 2013.

[4]Cary, E. and Smith, D., “A Modular, Data Driven System: Architecture for GSFC Ground Systems: GSFC’s Mission Services Evolution Center (GMSEC),” Groudn system architecture workshop, manhattan beach, california, march 30- april 1, 2004.

[5]“SALOME, The Open Source Integration Platform for Numerical Simulations,” http://www.salome-platform.org, accessed dec 2013/.

[6]Ribes, A. and Caremoli, C., “Salome platform component model for numerical simulation,” in proceeding: Compsac ’07 proceedings of the 31st annual international computer software and applications conference - volume 02 pages 553-564, 2007.

[7]Schlüter, J. U., Wu, X., Shankaran, S., Kim, S., Pitsch, H., and Alonso, J., “A framework for coupling Reynolds-Averaged with large eddy simulations for gas turbine applications,,” ASME Journal of Fluids Engineering, 127, pp 806-815, 2005.

[8]Schlüter, J. U., Wu, X., v.d. Weide, E., Hahn, S., Herrmann, H., Alonso, J. J., and Pitch, H., “A python approach to multi-code simulations: CHIMPS,” Center for Turbulence Research, Annual Research Briefs 2005 .

[9]http://www.python.org/, last accessed Januray 2014.

[10]Stevens, R. W., *UNIX Network Programming, Volume 2, Second Edition: Interprocess Communications*, Prentice Hall, 1999.

[11]Hall, B., “Beej’s Guide to Network Programming Using Internet Sockets,” http://beej.us/guide/bgnet/, last accessed Dec 2013.

[12]Moore, K. T., Naylor, B. A., and Gray, J. S., “The Development of an Open Source Framework for Multidisciplinary Analysis & Optimization,” AIAA 2008-6069, 12th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, 10 - 12 September 2008, Victoria, British Columbia Canada.

[13]Perez, R. and Martins, J., “An Object-Oriented Framework for Aircraft Design Modeling and Multidisciplinary Optimization,” 12th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, Sept 10 2008.

[14]Hassan, D. and Ritter, M., “Assessment of the ONERA/DLR numerical aeroelastics prediction capabilities on the HIRE-NASD configuration,” IFASD-2011-109, 2011.

[15]Ghazlane, I., Carrier, G., Dumont, A., Marcelety, M., and Désidéri, J.-A., “Aerostructural Optimization with the Adjoint Method,” in Evolutionary And Deterministic Methods for Design, Optimization and Control, Eds. C. Poloni, D. Quagliarella, J. Périaux, N. Gauger and K. Giannakoglou, CIRA, Italy 2011.

[16]Morton, S., Lamberson, S., and McDaniel, D., “Static and Dynamic Aeroelastic Simulations Using Kestrel - A CREATE Aircraft Simulation Tool,” AIAA 2012-1800, 53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference, 23-26 April 2012, Honolulu, Hawaii.

[17]www.github.com, last accessed jan 2014.

[18]IEEE Standard Association, Standard Development Working Group, “POSIX - Austing Joint Working Group,” http://standards.ieee.org/develop/wg/POSIX.html, last accessed Januray 2014.

[19]The Open Group, "POSIX®1003.1 Frequently Asked Questions (FAQ Version 1.14)," http://www.opengroup.org/austin/papers/posix_faq.html, last accessed Januray 2014.

[20]The Open Group, "The Open Group Base Specifications Issue 7, IEEE Std 1003.1™, 2013 Edition," http://pubs.opengroup.org/onlinepubs/9699919799, last accessed Januray 2014.

[21]www.github.com/libm3l, last accessed jan 2014.

Appendix

libm3l basic functions

m3l_Add - adds a node to a list
m3l_Cat - prints the list content to standard output
m3l_Cp - copy a node to a different location
m3l_Fread - reads list from file
m3l_Fwrite - writes list to file
m3l_Ln - links a node to a specified position in list
m3l_Locate - locate a node in linked list
m3l_Mklist - make a node
m3l_Mv - move node to a different position in the list
m3l_Receive_send_tcpipsocket - receive list from the socket and send another list to the socket
m3l_Receive_tcpipsocket - receive list from the socket
m3l_Rm - removes a node from list
m3l_Send_receive_tcpipsocket - send list to the socket and receive another list from socket
m3l_Send_to_tcpipsocket - send list to the socket
m3l_Umount - delete entire linked list
m3l_cli_open_socket - client open socket
m3l_server_openbindlistensocket - server open, binds and listen socket

Sender code example

The Sender creates a linked list which contains a 1-dimensional array of 10 double numbers called *Pressure* with arbitrary values. In a loop it opens socket, sends the list to the Receiver through socket, closes socket and executes another round of the loop.

```
node_t *Gnode=NULL, *RecNode=NULL, *TmpNode = NULL;

lmchar_t *name="Pressure";
client_fce_struct_t InpPar, *PInpPar;
opts_t opts, *Popts_1;

nmax = 100000;
PInpPar = &InpPar;

for(i=0; i<nmax; i++){
/*
 * Create head node of the list with payload data
 */
Gnode = client_name("Text from Client1");

dim = (size_t *) malloc( 1* sizeof(size_t));
/*
 * add Pressure array, array has 10 pressure with some values
 */
dim[0] = 10;
if(  (TmpNode = m3l_Mklist("Pressure", "D", 1, dim, &Gnode, "/Client_Data", "./", (char *)NULL)) == 0)
Error("m3l_Mklist");
tmpdf = (double *)m3l_get_data_pointer(TmpNode);
for(j=0; j<dim[0]; j++)
tmpdf[j] = (i+1)*j*1.1;
free(dim);
/*
 * open socket
 */
PInpPar->data_name = name;
PInpPar->SR_MODE = 'S';
if ( (PInpPar->mode = get_exchange_channel_mode('D', 'N')) == -1)
Error("wrong client mode");
Popts_1 = &opts;
m3l_set_Send_receive_tcpipsocket(&Popts_1);

if( (sockfd = open_connection_to_server(argv[1], portno, PInpPar, Popts_1)) < 1)
Error("client_sender: Error when opening socket");
/*
 * send payload in Gnode to the socket
 */
client_sender(Gnode, sockfd,  PInpPar, (opts_t *)NULL, (opts_t *)NULL);
/*
 * close socket
 */
if( close(sockfd) == -1)
Perror("close");
/*
 * free borrowed memory
 */
if(m3l_Umount(&Gnode) != 1)
Perror("m3l_Umount");
  }
```

Receiver code example

The Receiver waits for data set identified by the name *Pressure*. Once the data set arrives, it prints the data set on stdoutput, locates a double array called *Pressure* and calculates the average value of the array.

```
char *name ="Pressure";
while(1){
/*
 * open socket
 */
PInpPar->data_name = name;
PInpPar->SR_MODE = 'R';
if ( (PInpPar->mode = get_exchange_channel_mode('D', 'N')) == -1)
Error("wrong client mode");

Popts_1 = &opts;
m3l_set_Send_receive_tcpipsocket(&Popts_1);

if( (sockfd = open_connection_to_server(argv[1], portno, PInpPar, Popts_1)) < 1)
Error("client_sender: Error when opening socket");
/*
 * receive data set from socket
 */
Gnode = client_receiver(sockfd, PInpPar, (opts_t *)NULL, (opts_t *)NULL);
/*
 * print received data set on stdoutput
 */
if(m3l_Cat(Gnode, "--all", "-P", "-L",  "*",   (char *)NULL) != 0)
Error("CatData");
/*
 * find Press data set in the List
 */
if( (SFounds = m3l_Locate(List, "/Client_Data/Pressure", "/*/*", (char *)NULL)) != NULL){

TmpNode = m3l_get_Found_node(SFounds, 0);  /* get pointer on the first Press data
                                                (in case there is more then
                                                 one Press data set) */
pmax = m3l_get_List_totdim(TmpNode);             /* get array dimension */
press_val = (double *)m3l_get_data_pointer(TmpNode);    /*get pointer on array with Press values */
/*
 * release borrowed memory from m3l_Locate
 */
m3l_DestroyFound(&SFounds);
/*
 * calculate average value
 */
ave = 0;

for(j=0; j<pmax;j++)
ave = ave + tmpdf[j];
printf("\n\n Average value of pressures is %.16lf\n", ave = ave/pmax);
}
else
{
printf(" No founds\n");
}
/*
 * free borrowed memory
 */
if(m3l_Umount(&Gnode) != 1)
Perror("m3l_Umount");
/*
 * close socket
 */
if( close(sockfd) == -1)
Perror("close");

  }
```