**Modeling Discrete-Event Systems Using DEVS (2012 Fall)**

**Assignment1: Offline File Transfer Simulator**

Weiwei Li 6835268 Ottawa University

Yuhan Zeng 6748530 Ottawa University

**Part I:**

Offline file transfer: Offline file transfer is used to support asynchronous file transfer between different user agents. This happens in the case that, for instance, user agent A tries to send a file to user agent B while user agent B is offline. In synchronous file transfer mode, User agent A must wait until user agent B is online. A substitute solution for this situation is to provide an offline file transfer service. The key concept for this server is that user agent A first sends the files to a temp exchange server and when user agent B is online, it could try to download the files from the temp exchange server. To fully utilize the temp exchange server, files will be deleted after successfully delivery to provide more capacity for other potential user agents.



The Simulator consists of 3 components: UA, network and UB.

The network can be divided into three parts: fileExchangeServer and 2 AccessNetworks. File exchange server is comprised of four modules: tmpFileReadThread, tmpFileWriteThread, fileServerSender and fileServerReceiver.

There are 2 subnets: AccessNetworkA and AccessNetworkB . Each subnet has two components: the sending subnet and receiving subnet.

We reuse the ABR module presented in simulation sample. With this module, we could provide reliable file transfer between user agents (UA and UB) and file exchange server. The UA sends a file to the receiving module of file exchange server through AccessNetwork A and waits for an acknowledgement. If the acknowledgement doesn't arrive within a predefined time, UA will resend this file until it receives an expected acknowledgement and then becomes passive. UB will receive the file from the sending module of file server through AccessNetwork B. The sender module will use the same protocol utilized in the communication between UA and the receiver of file server to ensure reliable file transfer.

File server have 2 different behaviors according to its sub modules. The fileServerReceiver module is to receive the file from UA through AccessNetwork A and transfer it to tmpFileWriteThread. Then the tmpFileWriteThread will write the file into the file system. The fileServerSender module receives a requestIn signal which means UB is activated and starts to send the files stored in the file system to UB through AccessNetwork B with the help of tmpFileReadThread. tmpFileReadThread will delete the files after successful delivery.

**Part II**

OfflineFileTransfer simulator has two inputs and three outputs which are shown in Figure 1. The controlIn input means the files that need to be transferred from user agent A (UA) to FileExchangeServer. The requestIn input means user agent B (UB) wants to download the files. The three outputs indicate an expected acknowledgement that has been received (ackReceived), total number of files that has been sent (fileSent) and the file name that UB has received (outData), respectively. OfflineFileTransfer simulator has three components: UA, UB and Network. Network can be further decomposed into three components which are AccessNetworkA, AccessnetworkB and FileExchangeServer. UA sends files to FileExchangeServer through AccessNetworkA. During or after the time that UA sends the files, UB might request the files from FileExchangeServer via AccessNetworkB. FileExchangeServer then sends the files to UB and deletes the files one by one once UB has successfully received the files. The FileExchangeServer can also be further decomposed into four components and they are fileServerReceiver, tmpFileWriteThread, tmpFileReadThread and fileServerSender. fileServerReceiver receives the files from UA and sends the files to tmpFileWriteThread. Then tmpFileWriteThread will write the files into file system (emulated by a queue). After receiving the request from UB, fileServerSender will request the files from tmpFileReadThread and notify it to delete the files after successful delivery. AccessNetworkA and AcessNetworkB have the same structure and could also be further decomposed into two subnets. The two subnets have the same behaviors which also use the atomic model called subnet.

OfflineFileTransfer has six atomic models: sender, receiver, subnet, tmpFileWriteThread, tmpFileReadThread, fileServerSender and three of them (sender, receiver and subnet) are reused models written by Tao Zheng. We made some modifications to these models to fix some bugs together with some bad code convention. We wrote these kinds of information into the header of the files that we modified.

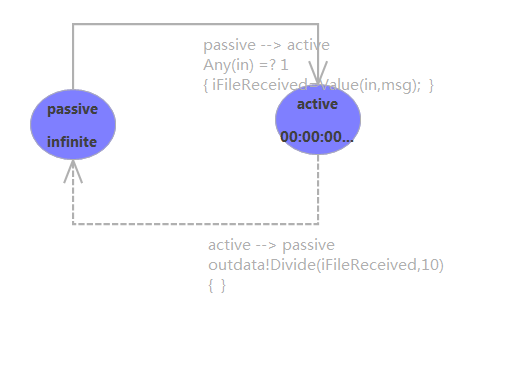
**Atomic Modle:**

Formal Specification:

The formal specification for the atomic models is as follows:

Atomic DEVS=<S, X, Y, δint, δext, λ, ta >

**Receiver/UB :( share the same atomic model)**



S= {passive, active};

X= {in};

Y= {out,outData}

δint (active)= idle;

δext(in,passive)= active;

λ (active)

{

// offset is a pre-defined constant with value of 10

send in / offset to port outData; // extract data

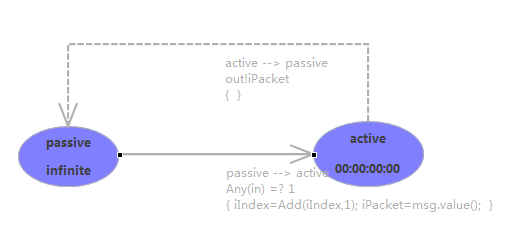
send in % offset to port out; // extract alternative bit

}

ta (idle)= INFINITY;

ta (active) = 10ms;

**Subnet:**



S= {passive, active};

X= {in};

Y= {out};

δint (active)= idle;

δext(in,passive)= active;

λ (active)

{

send data from port in to port out // 95% chance to send the file

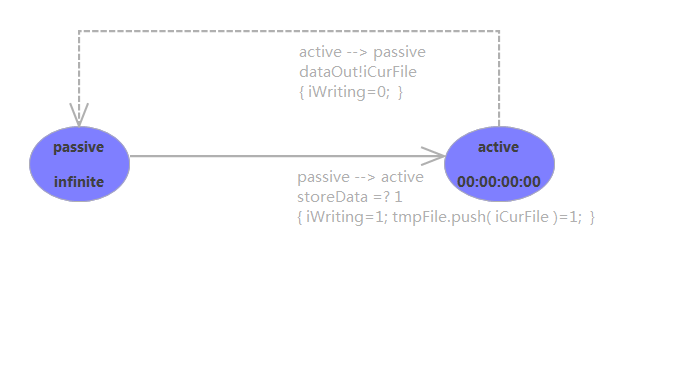
send nothing to port out // 5% chance to lose the file

}

ta (passive)= INFINITY;

ta (active)= delay; //normal distribution

**tmpFileWriteThread**



State Variables:

iCurFile = -1; // Current received file

iWriting = false; // true, write the file to file system

tempFile; // File system handle

S= {passive, active};

X= {storeData};

Y= {dataOut};

δint (active)= idle;

δext (iCurFile, iWriting, tmpFile, idle, x)

{

case phase

passive:

if x is from storeData and and phase

{

iCurFile = x;

if iCurFile >0 and there is no iCurFile found in file system;

put iCurFile into the file system;

sigma=0;

phase = active;

}

else

{

do nothing;

}

active:

do nothing;

}

λ (iWriting & iCurFile > 0 &active)

{

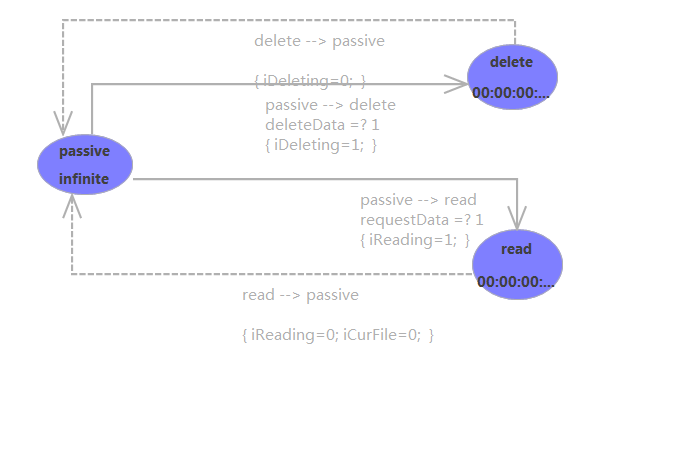
send iCurFile to port dataOut; // output file name

}

ta (passive)= INFINITY;

ta (active)= 0 ms;

**tmpFileReadThread**



State Variables:

iCurFile= -1;

iReading= false; // true, reading the file from file system

iDeleting= false; // true, deleting the file from file system

tempFile; // File system handle

fileNotFound = -1; // constant value

readyForRequest = 0; // constant value

fileSystemEmpty = -2; // constant value

S= {passive, delete, read};

X= {deleteData, requestData};

Y= {dataOut};

δint (delete)= passive;

δint (read)= passive;

δext (iCurFile, iReading, iDeleting, passive, x)

{

case phase:

passive:

if x is from requestData & t!tmpFile.empty()

{

iReading = true; // phase is changed to read

iCurFile = tmpFile.front();

}

if x is from deleteData & !tmpFile.empty()

{

iDeleting = true; // phase is changed to delete

tmpFile.pop();

iCurFile = tmpFile.front();

}

delete :

read:

do nothing;

}

λ (read&!tmpFile.empty())= iCurFile;

λ (read& tmpFile.empty())= fileNotFound;

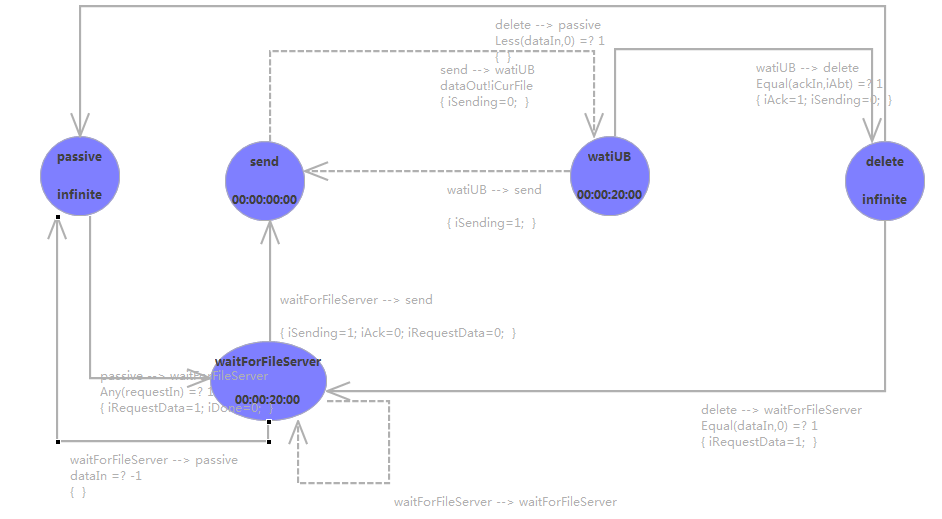
λ (delete &!tmpFile.empty())= readyForRequest;

λ (delete& tmpFile.empty())= fileSystemEmpty;

ta (passive)= INFINITY;

ta (active)= 0ms;

**fileServerSender**



State Variables:

iAck = false; // ack from UB

iCurFile = -1; // current sending file name

iAbt = 0; // alternative bit

iSending = false; // true if sending

iRequestData = false; // true if request data from file system

iDone = true; // false if files are being tranferred

S= { passive, send, waitForFileServer, delete, waitUB };

X= {requstIn,dataIn,ackIn};

Y= {requestData, ackReceived, fileSent, dataOut};

δint ((iAck, iSending, iRequestData, iDone)

{

Phase:

case delete:

{

if (x in dataIn & x == 0)

{

phase = waitForFileServer;

}

else if(x in dataIn & x<0)

{

phase = passive; // all files are transferred.

}

}

Case waitForFileServer:

{

If(x in dataIn & x>0)

{

phase = send;

}

else if( x in data & x <0)

{

phase=passive;

}

}

Case waitUB:

{

If(x in ackIn and a == iAbt)

{

iSending = false;

phase=delete;

}

}

}

δext(iAck, iSending, iRequestData, iDone, iCurFile, x)

{

case phase:

waitForFileServer:

{

Sigma = 20;

Phase = waitForFileServer;

}

send:

{

iSending = false;

sigma =0;

phase = waitUB;

}

waitUB:

{

iSending =true;

sigma = 20;

phase = send;

}

}

λ (iSending&active)=iCurFile & dataSent

λ (iDeleting&active )= ackReceived;

λ (iRequestData&active )= requestData;

ta (idle)= INFINITY;

ta (waitForServer)= 10ms;

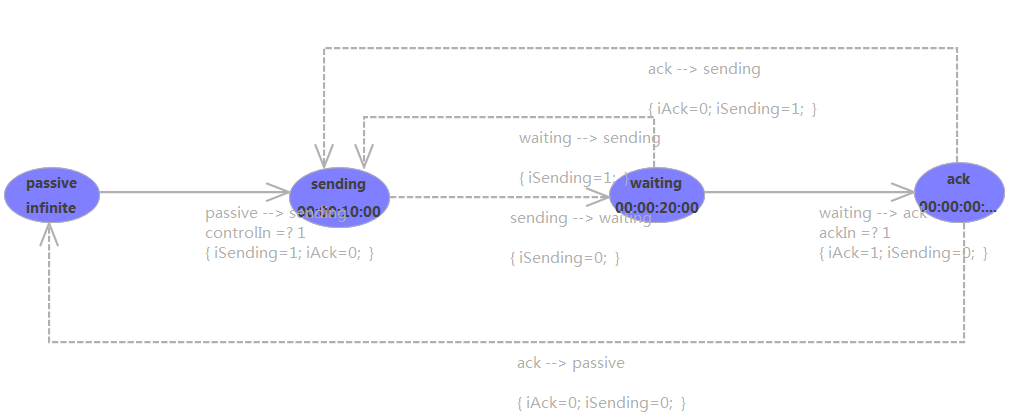
ta (send)=0;

ta (waitUB)=10ms;

ta (send)=0;

ta (delateData)=INFINITY;

**Sender**



State Variables:

iAck = false; // true if receive ack from receiver

iFileNum = 0;

iTotalFileNum = 0;

iAbt = 0;

iSending = false; // true if need to send file to receiver

S= {passive, sending,waiting,ack};

X= {controlIn,ackIn};

Y= {ackReceived, fileSent, dataOut};

δint (iAbt, iSending, iFileNum, iTotalFileNum, iAck)

{

case phase:

ack:

{

if( iFileNum < iTotalFileNum )

{

iFileNum ++;

iAck = false;

iAbt = ( iAbt + 1 ) % 2;

iSending = true;

sigma =10 // wait for 10 ms to output

phase = sending;

}

else

{

iAck = false;

iSending = false;

sigma=0;

phase = passive;

}

}

sending:

{

iSending = false;

sigma =20;// wait for 20 ms to output

phase = active;

}

Waiting:

{

iSending = true;

sigma =10; // wait for 10 ms to output

phase=active;

}

}

δext(iAbt, iSending, iFileNum, iTotalFileNum, iAck,x)

{

case phase:

passive:

if(x is from controlIn)

{

iTotalFileNum = x;

if(iTotalFileNum > 0)

{

iFileNum = 1;

iAck = false;

iSending = true; // prepare to send the files

// Set alternative bit

iAbt = ( iAbt + 1 ) % 2;

sigma =10;// wait for 10 ms to output

phase = sending;

}

waiting:

{

If x is from ackIn

{

If(iAbt == x)

{

iAck= true;

iSending = false;

sigma = 0; // immediate send the file

phase = ack;

}

}

}

}

λ (sending )=iFileNum &fileSent

λ (ack )= ackReceived;

ta (idle)= INFINITY;

ta (sending)= 10ms;

ta (waiting)=20ms;

ta (ack)=0;

**Coupled Model**

The formal specification for the atomic models is as follows:

CM=<X, Y, D, {}, I, Z, select>

**Network:**

X= {inAtoN, inBToN, requestIn};

Y= {outNToA, outNToB};

D= {AccessNetWorkA, AccessNetWorkB, FileExchangeServer};

I (AccessNetWorkA) =FileExchangeServer;

I (FileExchangeServer) = AccessNetWorkB;

I (AccessNetWorkB) = self;

Z (AccessNetWorkA) =FileExchangeServer;

Z (FileExchangeServer) = AccessNetWorkB;

Z (AccessNetWorkB) = self;

SELECT:

({AccessNetWorkA, AccessNetWorkB, FileExchangeServer}) = AccessNetWorkA;

({AccessNetWorkB, FileExchangeServer}) = FileExchangeServer;

**AccessNetWorkA**

X= {inAToS, inFToS};

Y= {outSToA, outSToF};

D= {subnetAToF, subnetFToA};

I (subnetAToF) =self;

I (subnetFToA) =self;

Z (subnetAToF) =self;

Z (subnetFToA) =self;

SELECT:

({ subnetAToF, subnetFToA })= subnetAToF

**AccessNetWorkB**

X= {inBToS, inFToS};

Y= {outSToB, outSToF};

D= {subnetBoF, subnetFToB};

I (subnetBToF) =self;

I (subnetFToB) =self;

Z (subnetBToF) =self;

Z (subnetFToB) =self;

SELECT:

({ subnetBoF, subnetFToB }}= subnetFToB

**FileExchangeServer:**

X= {inAToF, inBToF, requestIn};

Y= {outFToA, outFToB};

D= {tmpfilereadthread, tmpfilewritethread, fileserverreceiver, fileserversender };

I (fileserverreceiver) = tmpfilewritethread;

I (tmpfilereadthread) = fileserversender;

I (fileserversender) =self;

I (fileserversender) = tmpfilereadthread;

Z (fileserverreceiver) = tmpfilewritethread;

Z (tmpfilereadthread) = fileserversender;

Z (fileserversender) =self;

Z (fileserversender) = tmpfilereadthread;

SELECT:

({ fileserverreceiver , tmpfilewritethread })= fileserverreceiver

({ fileserversender, tmpfilereadthread })= fileserversender

**OfflineFileTransfer simulator**

X= {controlIn, requestIn };

Y= {fileSent, ackReceived, outData};

D= {UA, Network, UB};

I (UA) = Network;

I (Network) =UB;

I (UB) = Network;

I (Network) =UA;

Z (UA) = Network;

Z (Network) =UB;

Z (UB) = Network;

Z (Network) =UA;

SELECT:

({UA,Network,UB})=UA;

({ Network,UB })=Network;

## Test Strategies

We test each atomic model separately using the test strategy called “black box” and then put them together into couple models and test them step by step using the strategy called integration test.

**Part III**

We create several test inputs to every atomic model and coupled models to see whether the behaviors of our models accord with our expectation.

### Receiver/UB :( share the same atomic model)

The input of receiver is expected to be a positive integer. And it will extract the file from the input. The last digit is 0 or 1 which is alternative bit used to judge whether the correct file has been delivered while the other digits are considered as the file being transferred.

In this part we put 7 inputs (11, 20, 31, 31, 40, 40 and 51). UB will only receive one copy of the file at one time and if a new file comes within the receiving time, the old file being transferred will be discarded and the file will be received from the start point. The receiver.ev is as follows:

00:00:10:00 in 11

00:00:30:00 in 20

00:00:45:00 in 31

**00:00:52:00 in 31**

00:01:25:00 in 40

**00:01:35:00 in 40**

00:01:55:00 in 51

Due to the input that data 31 and 40 have been sent twice with a very short interval and thus the receiver does not have enough time to finish the file transfer, we can expect that the receiver or UB will discard the file being transferred and start to receive the file from the beginning. The receiver.out shows the expected results:

00:00:20:000 out 1

00:00:20:000 outdata 1

00:00:40:000 out 0

00:00:40:000 outdata 2

00:01:02:000 out 1

00:01:02:000 outdata 3

00:01:45:000 out 0

00:01:45:000 outdata 4

00:02:05:000 out 1

00:02:05:000 outdata 5

### tmpFileWriteThread

The input of tmpFileWriteThread is expected to be a positive integer. It will write this file into file system. However, if the file that tmpFileWriteThread receives has already exists in the file system. It will discard the file to gain a better performance.

In this part, we will store 3 data (1, 2, and 3). When there is an input, the tmpFileWriteThread should test whether the file has been stored in the file system to avoid unnecessary overwriting. In order to test this part, we put data 2 and data 3 three times without any sequence. The tmpFileWriteThread.ev is as follows:

00:00:10:00 storeData 1

00:00:12:00 storeData 2

00:00:14:00 storeData 2

00:00:15:10 storeData 3

00:00:16:20 storeData 3

00:00:17:30 storeData 2

00:00:18:00 storeData 3

Due to the input, we can expect that there will be only 3 outputs: 1, 2 and 3. And the expected time of each data will be the first time they are sent. The tmpFileWriteThread.out shows the expected results:

00:00:10:000 dataout 1

00:00:12:000 dataout 2

00:00:15:010 dataout 3

### Subnet

The input of subnet is expected to be a positive integer. Subnet will random delay the time of file being transferred. In addition, subnet only has 95% probability to successfully deliver the file which means the file might be lost in the subnet.

In this part, we put 20 inputs every 10ms with the alternate bits. The alternate bit is the last digit. There is no other condition in this part. The subnet.ev is as follows:

00:00:10:00 in 11

00:00:20:00 in 20

00:00:30:00 in 31

00:00:40:00 in 40

00:00:50:00 in 51

00:01:10:00 in 60

00:01:20:00 in 71

00:01:30:00 in 80

00:01:40:00 in 91

00:01:50:00 in 100

00:02:00:00 in 111

00:02:10:00 in 120

00:02:20:00 in 131

00:02:30:00 in 140

00:02:40:00 in 151

00:02:50:00 in 160

00:03:00:00 in 171

00:03:10:00 in 180

00:03:20:00 in 191

00:03:30:00 in 200

If not considering the file lost, the output will be expected to have the same sequence. However, as the file has 5% chance to be lost and from the two bold events shown below we could see one file has been lost. Additionally, because the time delay of the subnet is random, the expected time of the output will also be random. The subnet.out shows the expected results:

00:00:12:987 out 11

00:00:21:796 out 20

00:00:31:957 out 31

00:00:43:035 out 40

00:00:52:182 out 51

00:01:13:160 out 60

00:01:23:849 out 71

00:01:33:655 out 80

00:01:43:446 out 91

00:01:52:830 out 100

00:02:02:400 out 111

00:02:13:687 out 120

00:02:21:055 out 131

00:02:33:678 out 140

00:02:43:679 out 151

**00:02:54:759 out 160**

**00:03:12:533 out 180**

00:03:21:704 out 191

00:03:32:419 out 200

### tmpFileReadThread

tmpFileReadThread has two inputs. The inputs are expected to be an integer. The input of requestData will trigger tmpFileReadThread to read the file from the file system while delteData will trigger the deletion. tmpFileReadThread will return error code if there is no file in the file system. For reading operation, this error code will be -1 and for deleting operation, this error code will be -2.

In this part, we use requestData 20(20 has no actual meaning and could be replace by any number) to test the request part and use deleteData 20 to test the delete part. The time is random. The tmpFileReadThread.ev is as follows:

00:00:18:00 requestData 20

00:00:20:00 deleteData 20

00:00:21:00 requestData 20

00:00:22:00 requestData 20

00:00:24:00 requestData 20

00:00:26:00 deleteData 2

00:00:28:00 requestData 20

00:00:29:00 requestData 20

Because we test each atomic model separately, there is no data in the file system. Therefore, the expected dataout should always be -1 or -2, respectively. The tmpFileReadThread.out shows the expected results:

00:00:18:000 dataout -1

00:00:20:000 dataout -2

00:00:21:000 dataout -1

00:00:22:000 dataout -1

00:00:24:000 dataout -1

00:00:26:000 dataout -2

00:00:28:000 dataout -1

00:00:29:000 dataout -1

### fileServerSender

fileServerSender is the most complicated atomic model in this simulator. It has two handshaking communication with tmpFileReadThread. fileServerSender has three inputs and four outputs. The input of dataIn is expected to be an integer greater than -3. The input of ackIn is expected to be an integer of 0 or 1. The input of requestIn is expected to be an integer. The output of dataOut is expected to a positive integer. The output of ackreceived is expected to an integer. The output of requestData is expected to a positive integer of 1. The output of fileSent is expected to a positive integer. fileServerSender will be activated once it received an requestIn request. It will then send a requestData to tmpFileReadThread. tmpFileReadThread will then send an answer to fileServerSender. If the received number is -1 which means no file exists in the file system, fileServerSender will finish the transfer and go to passive state. If this number is a positive integer, fileServerSender will send this number to UB via AccessNetworkB and wait for acknowledge from UB. When fileServerSender receives acknowledge from UB, it will send ackreceived to tmpFileReadThread to delete the file just transferred and wait for tmpFileReadThread’s response. tmpFileReadThread will response fileServerSender’s request and send its output to dataIn of fileServerSender with as following principles. If the file is deleted successfully and there are more files in the file system, it will send code 0 to fileServerSender. If file is deleted successfully and no more file exists or the file system has already been empty, then tmpFileReadThread will send -2 to fileServerSender. Once fileServerSender receive this response, it will have two choices: first, finish the whole transfer if the response is -2; second, send requestData to fileServerReadThread if the response is 0. This process will recursively executed until there is no file existing in the file system. The tmpReadThread.ev is as follows:

00:00:00:00 requestIn 10

00:00:04:00 dataIn 1

00:00:16:00 ackIn 1

00:00:18:00 dataIn 2

00:00:46:00 ackIn 1

00:00:47:00 dataIn 3

00:00:58:00 ackIn 1

00:00:59:00 dataIn 4

00:01:13:00 ackIn 1

00:01:13:00 dataIn 5

00:02:13:00 ackIn 1

Due to the input we can expect that there will be there will be several dataOut from 1 to 5 with their own alternative bits. Additionally, there will be fileSent from 1 to 5 sharing the same time with dataOut. Also there will be ackreceived when tmpFileReadthread receives the data. However, the outputs shows some different situations which will never happen in the reality. In the output, between dataout 2 and dataout 3, there is no ackreceived. Same thing happens between dataout 4 and dataout 5. This is due to the reason that from the input, the expected ackIn of dataIn 2 is 1 rather than 0. In the reality, if fileSeverSender receives wrong ackIn or no ackIn, it will keep sending until it gets an expected ackIn. However, because we test the model separately, when the fileSeverSender keeps sending, another dataIn comes, which changes expected the alternate bit from 1 to 0 in the first case and thus stop fileSeverSender sending.

Therefore, although it may have some differences from the reality, the output can still prove the model to be correct. The fileServerSender.out is as follows:

00:00:00:000 requestdata 1

00:00:14:000 dataout 11

00:00:14:000 filesent 1

00:00:16:000 ackreceived 1

00:00:28:000 dataout 20

00:00:28:000 filesent 2

00:00:48:000 dataout 20

00:00:48:000 filesent 2

00:01:08:000 dataout 20

00:01:08:000 filesent 2

00:01:28:000 dataout 20

00:01:28:000 filesent 2

00:01:57:000 dataout 31

00:01:57:000 filesent 3

00:01:58:000 ackreceived 1

00:02:09:000 dataout 40

00:02:09:000 filesent 4

00:02:29:000 dataout 40

00:02:29:000 filesent 4

00:02:53:000 dataout 51

00:02:53:000 filesent 5

00:03:13:000 ackreceived 1

### UA (Sender)

Sender has two inputs and three outputs. The input of controlIn is expected to a positive integer. The input of ackIn is expected to a positive integer of 0 or 1. The Output of dataOut is expected to be a positive integer. The output of fileSent is expected to be a positive integer. The output of ackReceived is expected to be positive integer of 1. In this part, the controlIn contains the total file numbers called iTotalFileNum. When a controlIn comes the UA starts sending files to the Network. And when the UA receives an ackIn from the Network, it will check the alternative bit. If there is no ackIn or it is an unexpected ackIn, the UA will resend the file until it receives the expected ackIn. After that the iFileNum will add 1. If iFileNum is smaller than the iTotalFileNum, the UA will keep sending until iFileNum equals to iTotalFileNum. The sender.ev is as follows:

00:00:00:00 controlIn -1

00:00:05:00 controlIn 0

00:00:10:00 ackIn 0

00:00:15:00 controlIn 5

00:00:30:00 ackIn 1

00:01:30:00 ackIn 0

00:01:55:00 ackIn 1

00:02:20:00 ackIn 1

00:02:45:00 ackIn 0

00:02:50:00 controlIn 3

00:02:55:00 ackIn 1

From the input, we can expect that there will be no dataout in the first ten seconds because the total file number is not larger than 0. However, there also exits some differences from the reality. At the time of 00:02:20:00, there comes a wrong ackIn. Thus the UA resends the file and finally gets the right ackIn on 00:02:45:00. However, on 00:02:50:00, another controlIn comes which changes the iTotalFileNum into 3. But the iCurFileNum does not change. So after getting another ackIn, it will stop sending. Therefore, it still proves that the model is correct. The Sender.out is as follows:

00:00:25:000 dataout 11

00:00:25:000 filesent 1

00:00:30:000 ackreceived 1

00:00:40:000 dataout 20

00:00:40:000 filesent 2

00:01:10:000 dataout 20

00:01:10:000 filesent 2

00:01:30:000 ackreceived 0

00:01:40:000 dataout 31

00:01:40:000 filesent 3

00:01:55:000 ackreceived 1

00:02:05:000 dataout 40

00:02:05:000 filesent 4

00:02:35:000 dataout 40

00:02:35:000 filesent 4

00:02:45:000 ackreceived 0

00:02:55:000 ackreceived 1

### AccessNetwork

The accessNetwork has two inputs and two outputs. All the inputs are expected to be a positive integer. All the outputs are expected to an integer of 0 or 1.

The AccessNetwork is made up of two parts: subnetAToF and subnetFToA. The subnetAToF gets in1 from UA and sends out1 to fileServerReceiver. The subnetFToA gets in2 from the fileServerReceiver and sends out2 to UA. Therefore, there are two inputs and two outputs in this part. The last digit of the input is the alternate bit which is also what in2 should receive from the fileServerReceiver. The AccessNetwork.ev is as follows:

00:00:10:00 in1 11

00:00:15:00 in2 1

00:00:20:00 in1 20

00:00:25:00 in2 0

00:00:30:00 in1 31

00:00:35:00 in2 1

00:00:40:00 in1 40

00:00:45:00 in2 0

00:00:50:00 in1 51

00:00:55:00 in2 1

00:01:10:00 in1 60

00:01:15:00 in2 0

00:01:20:00 in1 71

00:01:25:00 in2 1

00:01:30:00 in1 80

00:01:35:00 in2 0

00:01:40:00 in1 91

00:01:45:00 in2 1

00:01:50:00 in1 100

00:01:55:00 in2 0

We can expect that the outputs from the test events above. Because the time is random, the out1 and out2 are not alternative. Also there is no mechanism to prevent file loss, so we can spot file loss from the output. The AccessNetwork.out shows the expected results:

00:00:12:987 out1 11

00:00:21:957 out1 20

00:00:28:035 out2 0

00:00:32:182 out1 31

00:00:38:160 out2 1

00:00:48:655 out2 0

00:00:53:446 out1 51

00:00:57:830 out2 1

00:01:12:400 out1 60

00:01:18:687 out2 0

00:01:21:055 out1 71

00:01:28:678 out2 1

00:01:33:679 out1 80

00:01:39:759 out2 0

00:01:43:084 out1 91

00:01:47:533 out2 1

00:01:51:704 out1 100

00:01:57:419 out2 0

### fileServer

In this part, there are 3 inputs (inAToF, inBToF, requestIn) and 2 outputs (outFToA, outFToB). We put 3 different values (11, 20, 31) into the inAToF. The last digit of the value is alternate bit. Also, we put some alternate bits into inBToF. In order to test the model, we intentionally extend the time between each alternate bits to inBToF.

The fileserver.ev is as follows:

00:00:10:00 inAToF 11

00:01:12:00 inAToF 20

00:02:14:00 inAToF 20

00:03:15:10 inAToF 20

00:04:16:20 inAToF 31

00:05:17:30 inAToF 31

00:06:18:00 requestIn 1

00:07:58:00 inBToF 1

00:08:40:00 inBToF 0

00:09:58:00 inBToF 1

00:10:20:00 inBToF 0

00:11:00:00 inBToF 1

00:12:20:00 inBToF 0

From the inAToF, we can expect that the outFToA will be the alternate bit of inAToF after ten seconds from sending files. Because without inBToF, the fileServerSender will keep sending files to UB every 30 seconds, we can expect that there will be many same outputs every 30 seconds until the fileServerSender receives a new acknowledgement from inBToF. In addition, when the fileServerSender receives an output from sunbnetBToF, it will delete the file. Thus after receiving the acknowledgement on 00:09:58:00, there will be no files anymore. Therefore, even if there are still inBToF comming, no output should be generated. The fileServer.out shows the expected results:

00:00:20:000 outftoa 1

00:01:22:000 outftoa 0

00:02:24:000 outftoa 0

00:03:25:010 outftoa 0

00:04:26:020 outftoa 1

00:05:27:030 outftoa 1

00:06:28:000 outftob 11

00:06:58:000 outftob 11

00:07:28:000 outftob 11

00:08:08:000 outftob 20

00:08:38:000 outftob 20

00:08:50:000 outftob 31

00:09:20:000 outftob 31

00:09:50:000 outftob 31

### Network

In this part, there are three inputs (inAtoN, inBtoN, requestIn), and two outputs (outntoa, outntob). When there is an incoming requestIn, the Network begins to generate the outputs. The Network.ev is as follows:

00:00:10:00 inAtoN 11

00:01:10:00 inAtoN 20

00:02:10:00 inAtoN 31

00:03:10:00 inAtoN 40

00:04:10:00 inAtoN 51

00:04:10:00 requestIn 1

00:05:10:00 inBToN 1

00:06:10:00 inBToN 0

00:07:10:00 inBToN 1

00:08:10:00 inBToN 0

00:09:10:00 inBToN 1

00:10:10:00 inBToN 0

00:11:10:00 inBToN 1

00:12:10:00 inBToN 0

00:13:10:00 inBToN 1

As fileExchangeServer module is a sub module of Network, thus the Network will keep sending outputs until it gets an acknowledgement. The Network.out shows the expected results:

00:00:24:783 outntoa 1

00:01:24:992 outntoa 0

00:02:25:342 outntoa 1

00:03:27:504 outntoa 0

00:04:22:830 outntob 11

00:04:25:846 outntoa 1

00:04:53:687 outntob 11

00:05:24:733 outntob 20

00:05:54:734 outntob 20

00:06:27:843 outntob 31

00:06:57:292 outntob 31

00:07:24:123 outntob 40

00:07:54:930 outntob 40

00:08:27:594 outntob 51

00:08:59:490 outntob 51

### offLineFileTransfer (the Top)

This is our final system test. The outLineFileTransfer is consisted of 3 parts. UA sends files to UB through Network and gets acknowledgement from the Network. To ensure the stability of the system, if one file lost, it will resent the file. In order to realize the offline transfer function, a requestIn port should be added to send the request to the fileExchangeServer to download the files sent by UA.

As described in part 1, there are two inputs (controlIn, requestIn) and three outputs (fileSent, ackReceived, outData). When a controlIn comes, the UA starts transmit files to the server. When the requestIn comes, the UB starts to get files from the server. The offLineFileTransfer.ev is as follows:

00:00:10:00 controlIn 20

00:05:00:00 requestIn 1

From above, we can expect that there are 20 files sent to UB and UB starts to get these files at 00:05:00:00. The offLineFileTransfer.out shows the expected results:

00:00:20:000 filesent 1

00:00:34:783 ackreceived 1

00:00:44:783 filesent 2

00:00:59:775 ackreceived 0

00:01:09:775 filesent 3

00:01:25:117 ackreceived 1

00:01:35:117 filesent 4

00:01:52:621 ackreceived 0

00:02:02:621 filesent 5

00:02:18:897 ackreceived 1

00:02:28:897 filesent 6

00:02:44:984 ackreceived 0

00:02:54:984 filesent 7

00:03:09:717 ackreceived 1

00:03:19:717 filesent 8

00:03:38:155 ackreceived 0

00:03:48:155 filesent 9

00:04:03:772 ackreceived 1

00:04:13:772 filesent 10

00:04:27:895 ackreceived 0

00:04:37:895 filesent 11

00:04:55:989 ackreceived 1

00:05:05:989 filesent 12

00:05:21:097 ackreceived 0

00:05:24:622 outdata 1

00:05:31:097 filesent 13

00:05:45:433 ackreceived 1

00:05:51:048 outdata 2

00:05:55:433 filesent 14

00:06:11:501 ackreceived 0

00:06:18:265 outdata 3

00:06:21:501 filesent 15

00:06:37:283 ackreceived 1

00:06:44:768 outdata 4

00:06:47:283 filesent 16

00:07:04:416 ackreceived 0

00:07:14:416 filesent 17

00:07:27:000 ackreceived 1

00:07:37:000 filesent 18

00:07:40:589 outdata 5

00:07:54:540 ackreceived 0

00:08:04:540 filesent 19

00:08:05:029 outdata 6

00:08:20:556 ackreceived 1

00:08:30:556 filesent 20

00:08:33:947 outdata 7

00:08:59:500 outdata 8

00:09:00:556 filesent 20

00:09:26:646 outdata 9

00:09:30:556 filesent 20

00:09:45:128 ackreceived 0

00:09:51:629 outdata 10

00:10:19:330 outdata 11

00:10:46:693 outdata 11

00:11:12:924 outdata 12

00:11:40:175 outdata 13

00:12:05:278 outdata 14

00:12:30:114 outdata 15

00:14:02:788 outdata 15

00:14:27:761 outdata 16

00:14:52:425 outdata 17

00:15:20:727 outdata 18

00:15:49:625 outdata 18

00:16:13:548 outdata 19

00:16:40:830 outdata 20

The offlineFileTransfer simulator simulates the process of offline file transfer and the result is just expected. The behaviors of each part including the coupled models and atomic models are tested separately. The transferred files are simplified as a sequence number in this model. Above all, from the simulation, we can spot that the offlineFileTransfer works quite well.