Deliverable: Assignment #2
Title: CS436: Distributed Computer Systems
Offering: Winter 2011

WWW: http://bit.ly/uw_446-11 [The web page will be periodically updated]
Twitter: @cs436 (http://twitter.com/cs436) [Important updates broadcast here]

Lectures: Tuesday & Thursday 0830 - 0950 MC 4064

Instructor: Dr. Reid Holmes; DC 3351. Office hours by appointment. rth.cs436@gmail.com
TAs: Ali Abedi; DC 3549. Office hours by appointment. a2abedi@cs.uwaterloo.ca

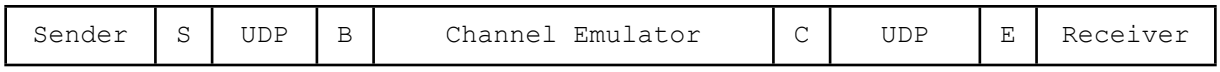
Due: 0830 on Mar 1 via email to rth.cs436@gmail.com

Notes:

This assignment can be completed individually or in a team of at most two people. Only one copy of the assignment needs to be submitted, but make sure that **both** team members names are on any submitted files. Your source code files / scripts / README should be be submitted as a zip file called <first-last>[_first-last]_a2.zip (if you are performing the assignment alone obviously the second first-last is not required). Do not submit any object or executable files. Email the assignment by 0830 on Mar 1 to rth.cs436@gmail.com. Late assignments will not be accepted.

Questions:

In this assignment, you are provided with a channel emulator for an unreliable channel. You must implement the selective repeat versions of reliable pipelined data transfer and a simple file transfer application. The reliable transfer protocol must be able to handle network errors such as packet loss, duplication, and reordering. For simplicity, the protocol only needs to be unidirectional, i.e. data flows in one direction only (from sender to receiver) and acknowledgements (`ACK`s) in the opposite direction. To implement this protocol, you need to write two programs: a sender and a receiver with the specifications given below. Communication to and from the channel emulator uses `UDP`. You can implement your solution in any programming language. The overall setup is shown in the schematic diagram below:



and uses the following addressing scheme (see Addressing for further information):

- `S`: address/port learned automatically by channel
- `B`: address/port written into 'channelInfo' by channel emulator, read by sender
- `C`: address/port learned automatically by receiver
- `E`: address/port written in 'recvInfo' by receiver, read by channel emulator

When the sender needs to send packets to the receiver, it sends them to the channel emulator at '`B`' instead of sending them directly to the receiver. The channel emulator then forwards the received packets to the receiver at '`E`'. However, it may randomly discard or delay packets. The receiver sends `ACKs` to the sender via the channel at '`C`', which may also randomly discard or delay `ACKs`.

Details:

Packet Format
All packets exchanged between the sender and the receiver must adhere to the following format:

Field	*Type*
`Packet Type`	32 bit unsigned integer, big endian
`Sequence Number`	32 bit unsigned integer, big endian
`Packet Length`	32 bit unsigned integer, big endian
`Payload`	byte sequence, maximum 500 bytes

The `Packet Type` field indicates the type of the packet. It is set to

- 0 if it is a data packet,
- 1 if it is an `ACK`, and
- 2 if it is an end-of-transfer (`EOT`) packet (see below for details).

For data packets, the `Sequence Number` is the modulo 32 sequence number of the packet, i.e. the sequence number range is [0,31]. For `ACK` packets, `Sequence Number` is the sequence number of the packet being acknowledged.

The `Packet Length` field specifies the total length of the packet in bytes, including the packet header. For `ACK` and `EOT` packets, the size of the packet is just the size of the header.

Sender Program
You must implement a sender program that takes two arguments: the value of a `timeout` in milliseconds and a `filename`. The sender then transfers the file reliably to the receiver program. The `timeout` is used as the timeout period for the reliable data transfer. During the transfer, the sender program should create packets as big as possible, i.e. containing 500 bytes payload, if enough data is available. After all contents of the file have been transmitted successfully to the receiver and the corresponding `ACK`s have been received, the sender should send an `EOT` packet to the receiver. The sender can close its connection and exit, after the receiver has responded with an `EOT` packet. You can assume that `EOT` packets are never lost. The sender must be robust in the presence of a faulty receiver. For example, when receiving unsolicited `ACK`s or `EOT`, the sender must report this as error to standard error, but otherwise continue.

Receiver Program
You must implement a receiver program. The receiver program takes one argument: the `filename` to which the transferred file is written. When the receiver program receives the `EOT` packet, it sends an `EOT` packet back and exits. The receiver must be robust in the presence of a faulty sender. For example, when receiving data packets out of range or an `EOT` while packets are still missing, the receiver must report this as error, but otherwise continue.

Output
For both testing and marking purposes, your sender and receiver program must print log messages to standard output - a line for each data packet being sent and received (including duplicates) in the following format:

```
PKT <SEND|RECV> {DAT,ACK,EOT} <sequence number> <total length>
```

For example:

```
PKT SEND DAT 17 512
PKT RECV ACK 17 12
```

You must follow this format to avoid problems during testing and marking.

Further, whenever your program executes a potentially blocking function call (read, recv, select, sleep, etc.), it must print a message to standard output, describing the call that is being made. The format of this message is not important, but it should not contain the string "PKT". If the program ever hangs, this output will help to determine why.

Selective Repeat
In the Selective Repeat variant, if there's data available for sending, the sender sends more data according to the current status of its send window. The send window size should be set to a fixed value of 10 packets. ACKs are not cumulative and only acknowledge the sequence number in the ACK packet. Follow the description of Selective Repeat as discussed in class.

Channel Emulator
The channel emulator is started with the following syntax:

```
channel <max delay> <discard probability> <random seed> <verbose>
```

All data and ACK packets are subject to a random delay, uniformly distributed between 0 and <max delay> milliseconds.

All data and ACK packets are subject to random discard with a probability of <discard probability>.

If <random seed> is set to a non-zero value, this seed is being used to initialize the random number generator. Multiple runs with the same seed produce the same channel behaviour. If <random seed> is set to zero, the random number generator is seeded with the current system time.

If <verbose> is set to a non-zero value, the channel emulator outputs information about its internal processing.

Addressing
In order to avoid global addressing settings, but enable quick testing, the following addressing scheme is used. The receiver program is started first and must write its 'E' address information (hostname and port number) into a file recvInfo that is read by the emulator. The channel emulator is started next and uses this information to send packets towards the receiver. The corresponding sending address 'C' of the emulator must be learned automatically by the receiver (see man recvfrom) upon reception of the first packet and used for sending all acknowledgements. The same mechanism is used between the sender and the emulator, i.e. the emulator writes its 'B' addressing information into a file channelInfo which is then read by the sender. The sending address 'S' of the sender is learned automatically by the channel. All files are read and written in the current directory. The contents of each file are the hostname and port number, separated by space. Example:

```
cpu06.student.cs 38548
```

Additional Comments/Hints
Since UDP is used for data transmission, delivery to and from the channel emulator is not guaranteed. This should not matter for data or ACK packets. You can ignore the residual probability that this could result in a loss of an EOT packet.

Be aware that the channel emulator only forwards two EOT packets (one in each direction) and then exits!

To implement the operating system timer using C/C++, you need to use either 'select' or 'ualarm' . Read the respective man pages to learn about these mechanisms.

In general, carefully read the man pages of all system calls that you are using.

Download
channel executable for Solaris (student.cs environment).
channel executable for Linux/x86 - unofficial.

Procedures

What to hand in
A zip file of all code files and a README file that describes which 'step' you managed to achieve. Your assignment must include a Makefile or clear compilation instructions in a 'README' file. The following executables (or start scripts) must be built or exist in the current directory. The sender program must accept a filename and a timeout in milliseconds as arguments. The receiver program must accept a filename as argument. The file specified at the sender must be transferred to the receiver and stored under the name given to the receiver.

The `<timeout>` parameter is passed to the reliable data transfer protocol.

```
sender <sendfilename> <timeout>
receiver <recvfilename>
```

Evaluation
The assignment is to be done individually or in pairs. Your program will be evaluated in the `student.cs` undergraduate computing environment. Your program should not silently crash under any circumstances. At the very least, all fatal errors should lead to an error message indicating the location in the source code before termination. Marks will be assigned as follows:

- Step 1: 20%
- Step 2: 20%
- Step 3: 20%
- Step 4: 20%
- Step 5: 20%

The requirements for each step are described in the posted notes for the lecture from Feb 3.

Make sure whatever you hand in at least works for the final step you have implemented. (e.g., if you were working on step 4 but ran out of time, make sure the step 3 requirements are met and still work). If you were partially done a step above your current step, include a description of what aspects of the next step you have completed, what parts remain, and what challenges prevented you from finishing; also, be sure your code is well documented in this scenario. We may assign part marks if significant progress has been made on the next step.

If the code cannot be run for the step you have completed, we cannot assign grades. Be absolutely sure you have validated your code on the `student.cs` environment and have named your scripts / programs as required. This is essential!