TÉCNICO LISBOA

Computer Networks — 2023/24, Q2		**Assignment:**	Project 1
Reliable File Transfer		**Version:**	1.0
Issued:	2023-12-07	**Submission Due:**	2023-12-15

1 Overview

In this project you will learn the basics of reliable data transfer using a simplified automatic repeat request (ARQ) approach. Towards this end we will be developing a simple file transfer system built on unreliable UDP messages. This system will include a client (acting as the file sender) that reads a file and sends it over to a remote server (the file receiver) that stores a new copy of it.

You will be given a basic implementation that simply splits the file into 1000 byte chunks, and sends them over the network while assuming no packet loss or reordering. A final chunk including less than 1000 bytes of file data (potentially even 0 bytes, if the file size is an exact multiple of 1000) is recognized by the receiver as the final chunk. Your job is to upgrade this code by implementing a sliding window ARQ algorithm, similar to what we learned in class, as per the specification below. This will allow the receiver to rebuild the file from the segmented data even when packets are lost or reordered.

Both the sender and receiver will keep track of the file transmission with a send and receive window, respectively, the size of which is specified as command line arguments. The send window, on the sender, keeps track of any outstanding chunks that have not yet been acknowledged. As the receiver acknowledges reception of these chunks, this window advances, allowing more chunks to be transmitted. The receive window, on the server, keeps track of the non-contiguous sequence of chunks that have been received but that still present gaps of not yet received data. As data received from the client fills in these gaps, the receive window also advances, allowing more progress to be made towards receiving the entire file.

The system is designed as a generalized form of ARQ, based on selective acknowledgments and timeout events. The sender retransmits data after a timeout or in response to duplicate acknowledgements, but it uses the selective acknowledgments to optimize the process and avoid retransmitting any data that actually made it through. This approach allows the system to implement any of the three common ARQ algorithms by carefully choosing the window sizes. Setting the send and receive window sizes to 1 implements a *stop-and-wait* algorithm. Making the send window larger than 1, while still keeping the receive window at 1 implements a *go-back-N* algorithm. Finally, setting both windows to sizes larger than 1 embodies a full *selective-repeat* algorithm.

Finally, this project will be evaluated with automated test scripts so it is important to follow the specification precisely. Points may be deducted if the project is not submitted as expected or if the build process does not work out of the box as described. It is also important to process inputs and outputs exactly as shown in the specification below.

Computer Networks — 2023/24, Q2		**Assignment:**	Project 1
Reliable File Transfer		**Version:**	1.0
Issued:	2023-12-07	**Submission Due:**	2023-12-15

2 Specification

2.1 The Sender

The sender executable must be called `file-sender` and takes exactly four arguments from the command line: the file to send, the server host name and port number, and the send window size. The send window size will never exceed 32. Invoking the sender will thus look something like this:

```
:~$ ./file-sender file.txt localhost 1234 5
```

With this information, the sender should create a UDP client socket and connect to the server at the designated host and port. Any error in parsing the arguments or creating the socket should result in an error, with an exit status of `EXIT_FAILURE`.

If all went well, the sender should immediately start segmenting the file and sending out the first round of chunks (1000 bytes per chunk, as many chunks as the send window size, one chunk per datagram). These datagrams should use the following format (defined in packet-format.h):

```
typedef struct __attribute__((__packed__)) data_pkt_t {
  uint32_t seq_num;
  char data[1000];
} data_pkt_t;
```

As the format suggests, the message contains a 32 bit sequence number (in network byte order), identifying the chunk starting from 0, followed by the chunk of actual file data, up to 1000 bytes. The size of the UDP datagram can be used to infer the amount of data enclosed. The final chunk in the file is recognized by having less than the full 1000 byte maximum chunk size. For example, a 500 byte file will have a single chunk: chunk 0, with 500 bytes. A 1000 byte file will have two chunks: chunk 0 with 1000 bytes and chunk 1 with 0 bytes.

After sending this series of chunks, the sender should wait for acknowledgments from the receiver. As these acknowledgments stream in, the send window should be updated. If a contiguous sequence of chunks at the base of the window are acknowledged, then the window should advance forward, allowing new chunks to be transmitted immediately. Any chunks that are (selectively) acknowledged out of order should be kept track of, so they will not be retransmitted later on. Any acknowledgment that does not advance the sender window should be seen as a duplicate acknowledgement, leading to the immediate retransmission of the first chunk in the window for a quick recovery. Any datagrams received from sources other than the receiver should be ignored.

While waiting for acknowledgments, the sender should timeout after 1 second. If 1 second has elapsed without any acknowledgments having been received, the sender should retransmit all unacknowledged data in the send window.

The sender should continue this procedure until all chunk have been acknowledged, at which point it terminates with exit status `EXIT_SUCCESS`. If three timeout events occur in succession (with no acknowledgments received in the meantime) then the sender should give up and terminate with exit status `EXIT_FAILURE`.

2.2 The Receiver

The receiver executable must be called `file-receiver` and takes exactly three arguments from the command line: the file to write to, the server port number, and the receive window size. The receive window size will never exceed 32. Invoking the receiver will thus look something like this:

```
:~$ ./file-receiver file.txt 1234 5
```

With this information, the receiver should create a UDP server socket and wait for incoming data from the sender. Any error in parsing the arguments or creating the socket should result in an error, with an exit status of `EXIT_FAILURE`. Once the first datagram is received from the sender, the receiver should ignore any subsequent datagrams from other sources.

As data chunks stream in, the receiver should keep track of the receive window and send out acknowledgment datagrams. If a contiguous sequence of chunks at the base of the window are received, then the window should advance forward, opening up a new set of chunks that the receiver will now accept. Any chunks that are received out of order should be kept track of and selectively acknowledged. Chunks that are received beyond the limits of the receive window should not be stored.

The receiver should immediately acknowledge any chunks, as soon as they are received. These acknowledgement datagrams should use the following format (defined in packet-format.h):

```
typedef struct __attribute__((__packed__)) ack_pkt_t {
  uint32_t seq_num;
  uint32_t selective_acks;
} ack_pkt_t;
```

As the format suggests, the acknowledgement message largely mirrors the receive window state. The first field is a 32 bit integer (in network byte order order) which indicates the chunk sequence number of the base of the receive window. The second field is a 32 bit mask (also in network byte order), representing the selectively acknowledged chunks, starting one position ahead of the receive window base (as the chunk at the base itself can never have been received, as that would advance the window).

The receiver should continue this procedure, reconstructing the received file chunk by chunk. Like the sender, the receiver also has a timeout, but in this case it uses it to

terminate. The receiver should wait 4 seconds for new packets (enough time for the sender to retransmit 3 times). If, upon timeout, the file is as of yet incomplete, the receiver should delete the partially received file and terminate with exit status `EXIT_FAILURE`. If, however, all chunks have been received and acknowledged, the receiver should simply terminate with exit status `EXIT_SUCCESS`.

2.3 Build Process

The project should be built using GNU Make and the corresponding Makefile should be submitted alongside the code accordingly. Though how you organize your code and build the project is up to you, both the sender and the receiver should be built with a simple call to `make` without any arguments. Once `make` is run in this fashion, both the `file-sender` and `file-receiver` binaries should be available within the same directory as the Makefile. The following Makefile does a basic build as specified above:

```
TARGETS = file-sender file-receiver

CC = gcc
CFLAGS = -Wall -O0 -g

default: $(TARGETS)

%: %.c
    $(CC) $(CFLAGS) -o $@ $<

clean:
    rm -f $(TARGETS)
```

2.4 Submission

Project development and submission will be handled via an online git repository that will be assigned to you on gitlab.rnl.tecnico.ulisboa.pt. As you develop the project, maintain the project C code and the Makefile used to build it in your git repository. Please do not commit any build artifacts including any object files or binaries that you built yourself. The submitted files, particularly the Makefile, should be placed in the base directory within the git repository, not within any subdirectory. Once you are ready to submit the final revision of your code, tag it as `project1-submission` and push the tag to the remote repo. This can typically be done as follows:

```
:~$ git tag project1-submission
:~$ git push origin project1-submission
```

Downloading and building the submission from the command line should look something

like the following:

```
:~$ mkdir test
:~$ cd test/
:~/test$ git clone <repo URL> .
:~/test$ git checkout project1-submission
:~/test$ ls
Makefile  file-receiver.c  file-sender.c
:~/test$ make
gcc -Wall -O0 -g -o file-sender file-sender.c
gcc -Wall -O0 -g -o file-receiver file-receiver.c
:~/test$ ls
Makefile  file-receiver.c  file-receiver  file-sender.c  file-sender
```

3 Advice

3.1 Storing Chunks

Although you will still need to implement the complete ARQ algorithm with a send and receive window, it might help to use the files themselves as storage for the actual contents of the file chunks. This means that, if convenient, and rather than loading and storing chunks temporarily in memory, you can perform direct random access reads and writes to and from the files at both the sender and receiver.

This can be done by using `fseek` or similar API calls. This way, when retransmitting a chunk, its contents can be directly reread from the file at the sender. Conversely, the receiver can also write to the output file, using `fseek` to fill in chunks regardless of order. Ultimately, once all chunks have been received correctly, and assuming the technique has been used correctly, the file will have all of the content in place. The end result should be a correctly reconstructed output file which is identical to the input file from the sender.

3.2 Managing Timeouts

There are several mechanisms for handling timeout events, from the `alarm()` function to busy waiting. Sockets, however, offer an easy mechanism by allowing the developer to set a timeout for receiving data. This can be configured using the `setsockopt()`, as follows:

```
int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
struct timeval tv;
tv.tv_sec = ...;
```

```
tv.tv_usec = ...;
setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));
```

Once configured in this way, the `recv()` and `recvfrom()` functions will block for the prescribed time for data to arrive. If data does not arrive by then, these functions return prematurely, returning `-1` to indicate an error.

3.3 Bit Vectors

The `selective_acks` field can be seen as a bit vector represented in binary logic indicating which chunks have been received and which have not. Manipulating such bit vectors can be done easily with careful use of bit shifts and bit-wise Boolean operations.

Setting a specific bit to `1` can be done by ORing a variable with a bit mask with all `0`s and a single `1` at the desired location:

```
var = var | (1<<N); // Set Nth bit to 1 in var.
```

Setting a specific bit to `0`, conversely, can be done by ANDing a variable with a bit mask with all `1`s and a single `0` at the desired location:

```
var = var & ~(1<<N); // Set Nth bit to 0 in var.
```

3.4 Debugging

The easiest way to debug a system is to output its state at key places in the code. In this project both standard output and standard error are ignored in both the sender and receiver. Feel free to use either or both of these channels as debug outputs.

Developing fault-tolerant systems can pose a challenge as many fault patterns occur only infrequently and inconsistently. As the fault pattern can be seen as a kind of input into the fault-tolerant system, one needs to find a way to programmatically feed it in as an input while testing and debugging. Such techniques are known as fault injection.

One way to perform fault injection is to shim in mock implementations of basic OS APIs that allow the developer to inject faults on demand. For example, one can implement calls to `sendto` and `recvfrom` that drop, delay, or corrupt packets or that enable traffic logging. For external API calls such as these this can even be done transparently using the `LD_PRELOAD` environment variable. This variable allows developers to hook into the dynamic symbol loader as an executable is loaded and to replace calls to external dynamically loaded functions with calls into a new mock library.

A simple mock library for the `sendto` and `recvfrom` calls is provided (log-packets.c) to help with debugging. This file has to be compiled into a shared object file for dynamic symbol loading, using a command like this:

```
:~$ gcc -shared -fPIC -o log-packets.so log-packets.c -ldl
```

Once this shared object file is ready, you can load it as a mock library by setting the `LD_PRELOAD` variable, like this:

```
:~$ LD_PRELOAD="./log-packets.so" ./executable args
```

If you look at the log-packets.c source, you'll notice that the library supports a variety of useful features, including delaying, dropping, and logging packets. This can all be configured with additional environment variables, as follows:

```
:~$ LD_PRELOAD="./log-packets.so" \
    SEND_DELAY="500" \
    DROP_PATTERN="01" \
    PACKET_LOG="sender-packets.log" \
    ./file-sender ...
```

This command runs the file-sender with the mock library in place. Each packet will be delayed by 500ms and the second packet sent will be discarded, while all others go through. All calls to `sendto` and `recvfrom` are logged to sender-packets.log, including the packets that are discarded so we can track the injected faults.

The resulting logs can be further processed to illustrate the message exchange as a message sequence chart (MSC). A simple script that generates an MSC diagram from these logs (generate-msc.sh) is also provided to help with debugging. This script takes as arguments an output EPS file and one or more log files:

```
:~$ ./generate-msc.sh msc.eps sender-packets.log
```

The output EPS file not only shows the sequence of events, as they occurred, but also shows the timings and annotates each message with the sequence number (the first 32 bits in the message, as discussed above). The accompanying script `run.sh` illustrates this pipeline with a simple example. Feel free to modify these tools as necessary for your own purposes.

Another challenge while developing networked software is the need for multiple machines to form a network. One way around this is network emulation and virtualization, as we have already been doing in class with `CORE`. Although you should still test your project using these tools, this system is simple enough that it can be tested on a single host, using the loopback interface. By simply having the sender connect to `localhost`, you can run both the sender and receiver on different terminal windows, as opposed to a more elaborate virtualized environment.