CS3102 Practical 1

Overview

The requirement of this practical was to implement a client-server network application which streams data from a stored audio file from the server to the client. The packets had to be transmitted using UDP at the transport level, and either stored as a wave file or streamed in real time on the client side using a application-level protocol.
The application-level protocol was to be designed to cope with varying levels of network disruption, including delay, jitter, packet loss and packet reordering. The application was testing against these factors using the netem tool within a shell script.

Design and implementation

I wrote my server and client applications in Python, using the 'socket' library to send packets over the network using UDP. I began to implement the application-level protocol using stop-and-wait with a large number of packets sent at a time, but switched to implementing a go-back-N protocol as I realised that should be more time efficient.

I ran into some early time efficiency problems because of excessive string manipulation, but brought the time to transfer a 5 minute song down to about 2 seconds over a connection with no loss. Once I started to implement the go-back-N protocol I slowed down the transfer however to prevent the rate of packets overflowing the buffer and also to allow the audio streaming to be testing.

I used multithreading in my client and server applications to allow sending packets while simultaneously listening for packets. I also implemented a queue on the client side to store the sorted packets as they came in from the server, and a further queue to play the packets through an audio stream at the appropriate rate.

I used very small waits between sending packets and acknowledgements to control the flow of packets and try to avoid buffer overflow. This was effective but tedious to find the right timings, and probably could have been achieved better with the use of queues on the server as well as client side, and possibly locking to better synchronise the threads on both server and client side.

Testing

Initially I used the 'ssh' tool to log into another lab machine remotely, and used some random number generation to emulate and then try to fix a level of packet loss. I then edited the client and server shell scripts to work with my programs and to connect to a remote host, and also

edited my server to work in the way that the shell scripts expect by serving multiple requests. By manually calling the ./server.sh and ./client.sh commands, the server can serve multiple requests from the client for different audio files and streaming or storing the file. However the testing.sh file occasionally stops after the first request, and without access to print-outs or logs from the server I was unable to debug this. Mostly it works for both requests however.

Evidence of multiple requests:

```
dgk2@pc2-140-l:~/Documents/cs3102/Practicals/Practical1/scripts $ ssh pc2-134-l.cs.st-andrews.ac.uk
Systems documentation can be found at https://systems.wiki.cs.st-andrews.ac.uk/
Please report any issues or send requests to fixit@cs.st-andrews.ac.uk
(When reporting a problem please include the hostname of the machine being used.)
This machine is running Fedora 29
Last login: Sat Mar  2 00:52:43 2019 from 138.251.29.150
dgk2@pc2-134-l:~ $ cd Documents/cs3102/Practicals/Practical1/scripts/
dgk2@pc2-134-l:~/Documents/cs3102/Practicals/Practical1/scripts $ ./server.sh
25132
../audio/Beastie_Boys_-_Now_Get_Busy.wav
ack1
ack2
ack3
100
200
300
400
500
600
700
800
900
1000
```

```
24200
24300
24400
24500
24500
24600
24700
24800
24900
25000
25100
Thread exited
```

```
Thread exited
25132
../audio/Beastie_Boys_-_Now_Get_Busy.wav
ack1
ack2
ack3
0
100
100
100
```

Despite the testing.sh script not always completing the second request, it is functional for the first request and can be used with the ‘netm’ tool to simulate various network conditions. I tested each condition with three strengths and graded the level of clarity with which my

program was able to deliver the song. The conditions can be retested by running the testing.sh script and editing the NETEM_PARAMS variable. If the terminal is closed while the testing script is still running, this can cause with the server not relinquishing the address. The easiest solution to this is to change the port number in the client, server and testing scripts.

	Minor	**Medium**	**Significant**
Loss	loss 0.1%	loss 1%	loss 5%
	No noticeable error	No quality deterioration, occasional skips	Frequent skips, more towards the start of the song
Delay	delay 5ms	delay 50ms	delay 250ms
	No noticeable error	Very occasional skips (1 or 2 within song)	Very occasional skips, first second mostly skipped
Jitter	delay 30ms 25ms	delay 30ms 15ms	delay 60ms 15ms
	Playback begins near normal speed but then slows drastically. Each sound frame is very spaced out.	Playback severely slowed, barely audible.	Doesn't appear to play at all. Occasional timeout errors.
Reorder	reorder 1% 5%	reorder 5% 10%	reorder 10% 20%
	Slight deterioration in quality	More severe quality deterioration, some skipping	Similar quality deterioration, lots more skipping

Evaluation

My program coped reasonably well with loss, with lower percentages of loss being almost entirely accounted for. Larger loss percentages caused failures in the application protocol, leading to timeouts and data being skipped.

Constant delay wasn't a particular issue for the program as the frequency of packets sending remained the same, but just delayed. This had the most pronounced effect on the very start of the song.

Jitter caused my program to perform very poorly, largely because of the design decision to use micro sleeps to control packet slow and to somewhat synchronise threads. Even small variations in the delay this played havoc with my program's protocol.

Reordering caused a fairly noticeable deterioration in quality, and some skipping as the go-back-N protocol had to resend more packets the more the order was changed.

Conclusion

My application achieved all of the basic functionality required in the specification, and was edited to work for the testing scripts. I have tested the performance of the program with varying levels of delay, jitter, loss and packet reordering, and have presented and analysed the results above. If I was to attempt the practical again I would make more use of queues as buffers, and attempt to synchronise threads in a more jitter-resistant way. Because of the difficulties of doing this in Python and the awkwardness of using timeouts I would probably reattempt the practical in Java.

Note- I have had to remove some of the wave files from the audio folder to get the zip file small enough to upload to MMS.