Peyman Shobeiri

Computer networks- PA 4

McGill university

ID: 261286438

Winter 2025

In this assignment, we need to create a system centered around UDP multicast to distribute files from a single sender to multiple receivers. The sender transmits data packets using a multicast group address 239.0.0.1, with the specified port set to 5000. Each receiver joins the multicast group to receive the data. Since UDP does not provide reliability or delivery guarantees, we have implemented file chunking, which involves splitting each file into fixed-size chunks of 8 KB. This approach simplifies handling and allows for localized corruption checks. Additionally, we have included a per-chunk checksum to independently verify the integrity of the data, ensuring it is received by the receiver without corruption. Receivers can explicitly request any missing or corrupted chunks through NACK-based retransmission and ask the sender to resend the data. The sender operates in a While loop to continuously broadcast all chunks, allowing late joiners or restarts to obtain missing data over time. This combination of looping and NACK requests ensures that each receiver can eventually create a complete copy of every file, even in the presence of packet loss or late arrivals.

In this project, we used a function called checksum_method to calculate the final checksum for packets. Specifically, each chunk on the sender's side is scanned byte by byte, summing the values and taking the least significant 8 bits of the result. The checksum is then stored in the chunk header. Upon receiving the chunk, the receiver recomputes the same sum of bytes. If the two checksums match, the chunk is considered correct. However, if they differ the receiver dumps the chunk and sends a Negative Acknowledgment (NACK), asking for a retransmission of that specific chunk. This approach allows us to resend only a single chunk of data in case of an issue, instead of the entire file.

Data Flow and Structures

The sender begins by initializing a UDP socket and configuring the multicast settings. It then loads the files from the disk and splits each file into 8 KB blocks. For each chunk, it computes a checksum and stores the data along with important metadata, such as the file ID, chunk ID, and the total number of chunks. In its main loop, the sender loops through each chunk of the files, sending them sequentially and polling for incoming NACK messages after each transmission. If a NACK is received, the corresponding chunk is immediately retransmitted. If no NACK is received, the system proceeds as normal, ensuring that the chunks remain available in subsequent loops for any late joiners.

The receiver joins the multicast session by binding to the same UDP port and subscribing to the multicast group address. When it receives each data chunk, it verifies the chunk's integrity using a checksum. If the checksum is correct, the chunk is stored in memory according to its index. If the checksum is incorrect, the receiver sends a NACK to request a retransmission. We have used NACK in our design which removes the redundant response from the receivers (ACKs overhead) so now the receivers only send a message if something is wrong or missing. By doing this we have the minimum number of control packets while ensuring full reliability over the UDP protocol. Also, sometimes the receiver scans its storage for any missing or corrupted chunks and sends

NACKs for any detected lost packets. Once all chunks of a particular file have been correctly received, they are written sequentially to disk. The code then creates an output file that matches the original file's name and puts the files there.

How do we handle late joiners, crashes, and corruption?

The sender continuously loops through data chunks that would allow late joiners to collect subsequent chunks in real time. They can use NACK to request any missing segments, asking the sender to retransmit those specific chunks. Additionally, if either the sender or receiver crashes temporarily, by restarting them they can recover any missed data which ensures complete file transfer as long as the sender continues running. By creating the system this way we could have minimal overhead since we only send NACK from the receiver to the sender in case of any missing or corrupted chunks while in the other part, the sender is continuously broadcasting the remaining data.

Collaboration

This project was completed individually, and I implemented all the code myself. I also tested the system using different test cases, and the results were correct.