CS 536 Spring 2024

Lab 5: Congestion Control for Audio Streaming

Team Members: Alireza Lotfi & Shahab Rahimirad

Contents

1 Collaboration Details

1.1 Implementation

1.1.1 Alireza

- concurrency_utils.c & concurrency_utils.h
- socket_utils.c & socket_utils.h
- MakeFile

From the main files, Alireza mainly focused on building the core code including preparing the packets, sending feedbacks from client and handling them in the server.

1.1.2 Shahab

- concurrency_control.c & concurrency_control.c
- queue.c & queue.h

From the main files, Shahab mainly focused on adding the important functionalities to the code including the addition of timers, signal handlers, playback, and the addition of logging mechanism.

1.2 Performance Evaluation

For this section, both members were physically present at HAAS to test the system's functionality. We needed to verify if sound transmitted from the server could be heard on the computers. Various scenarios were tested, as outlined in the results section of the report.

1.3 Writing

Both members contributed equally to this section, collaborating on Overleaf simultaneously. Alireza primarily focused on crafting the overview, while Shahab took the lead on developing the results section.

2 Code Overview

2.1 Server Side (`audiostreams.c`)

2.1.1 Overview

The server-side application implements a UDP-based audio streaming server, utilizing non-blocking I/O for concurrent client handling. It dynamically adjusts its sending rate based on buffer occupancy feedback, ensuring smooth playback under varying network conditions.

2.1.2 Key Features

- Multi-threaded architecture for concurrent client connections.
- Non-blocking I/O for responsive socket operations.
- Adaptive rate control based on buffer feedback.
- Efficient congestion control algorithms.

2.2 Client Side (`audiostreamc.c`)

2.2.1 Overview

The client-side application receives audio data from the server via UDP and plays it in real-time. It provides buffer occupancy feedback to the server for rate adjustment, ensuring optimal streaming quality.

2.2.2 Key Features

- Periodic timer and signal handler for audio playback and buffer management.
- Error handling mechanisms for network issues.

- ALSA library integration for audio device interaction.
- Semaphore-based synchronization for buffer access.
- Logging functionality for performance monitoring.

2.3 Concurrency Utilities (`concurrency_utils.c`)

2.3.1 Functionality

- `create_semaphore(const char* sem_name)`: Creates a semaphore with the specified name or recreates it if it already exists. It ensures exclusive access to shared resources.
- `handle_received_data(Queue* buffer, uint8_t* block, int num_bytes_received, sem_t* buffer_sem, int buffersize)`: Handles received data by enqueueing it into the buffer while ensuring mutual exclusion using a semaphore.

2.3.2 Implementation

The `create_semaphore()` function creates or recreates a semaphore to provide synchronization between concurrent processes or threads. It uses the `sem_open()` function with appropriate flags to create or open the semaphore. If the semaphore already exists, it is unlinked and recreated to ensure a clean state.

The `handle_received_data()` function manages the reception of data packets. It employs semaphore-based synchronization to prevent race conditions when modifying the buffer. Upon receiving data, it acquires the semaphore using `sem_wait()` to ensure exclusive access. If the buffer is about to overflow, it handles the overflow situation and releases the semaphore. Otherwise, it enqueues the received data into the buffer and releases the semaphore.

2.4 Congestion Control and Feedback Mechanism (`congestion_control.c`)

The congestion control module encompasses functions responsible for adjusting the sending rate and preparing feedback packets based on buffer occupancy. Additionally, it includes utility functions for calculating expected packet numbers and file size.

2.4.1 Functionality

- `adjust_sending_rate(double lambda, double epsilon, double gamma, double beta, unsigned short bufferstate)`: Adjusts the sending rate `lambda` based on feedback received from the client. The adjustment is calculated using either method D or method C, determined by the `CONTROLLAW` constant.

- `prepare_feedback(int Q_t, int targetbuf, int buffersize)`: Prepares feedback packets based on the current buffer state (`Q_t`), target buffer size (`targetbuf`), and total buffer size (`buffersize`). The feedback packet indicates the level of buffer occupancy to the server for congestion control.

- `get_file_size(FILE *file)`: Retrieves the size of the given file by seeking to the end of the file and determining the byte offset. It returns the file size in bytes.

- `calculate_expected_number_of_packets(const char *filename, int blocksize)`: Calculates the expected number of packets required to transmit the file specified by `filename`, considering the given `blocksize` for packetization. It calculates based on the file size and block size, ensuring all file data is transmitted.

2.4.2 Implementation

The `adjust_sending_rate()` function dynamically adjusts the sending rate of the server based on buffer state feedback from the client. It incorporates either method D or method C for congestion control, updating the sending rate accordingly.

The `prepare_feedback()` function calculates and generates feedback packets to inform the server about the current buffer occupancy. It computes the feedback value (`q`) based on the difference between the current buffer state and the target buffer size, ensuring smooth audio transmission.

Utility functions like `get_file_size()` and `calculate_expected_number_of_packets()` support congestion control by providing file-related information necessary for efficient audio streaming.

3 Results & Comparison

3.1 Lambda and Interval

In this section we will compare the changes in lambda and the interval between packets in the server as seen in Figure 3.1.

- **(Epsilon = 0.2, Beta = 0.3):** Lambda starts at 10 and quickly stabilizes to a value slightly above 3. The packet interval rapidly increases to around 250 and then stabilizes with minor fluctuations. Low epsilon and beta values suggest that changes are more conservative, leading to quick stabilization in both lambda and packet interval.
- **(Epsilon = 1, Beta = 0.5):** Lambda decreases steeply at the start and then oscillates with a decreasing amplitude over time. The higher epsilon and beta lead to a higher initial drop and greater oscillations. The packet interval has high volatility initially with sharp peaks and valleys but then enters a pattern of consistent fluctuations, indicative of a system struggling to find an equilibrium.
- **(Epsilon = 0.5, Beta = 0.1):** Lambda starts high, with fluctuations that decrease over time but maintain a consistent amplitude later on. The mid-range epsilon value suggests a balance between aggressiveness and conservativeness in adjustments. The packet interval shows a significant initial spike, much like in Graph 2, but then settles to a lower stable value with less fluctuation compared to Graph 3, indicating a less reactive system due to the lower beta.
- **(Epsilon = 0.8, Beta = 0.2):** Lambda exhibits more volatility and higher peaks than in Graph 1, likely due to the higher epsilon which allows for more aggressive

adjustment. Despite a lower beta, the system does not stabilize as quickly or as smoothly as in Graph 1. The packet interval shows a large spike initially, reaching up to 2000 before gradually stabilizing at a higher variance than in Graph 1.

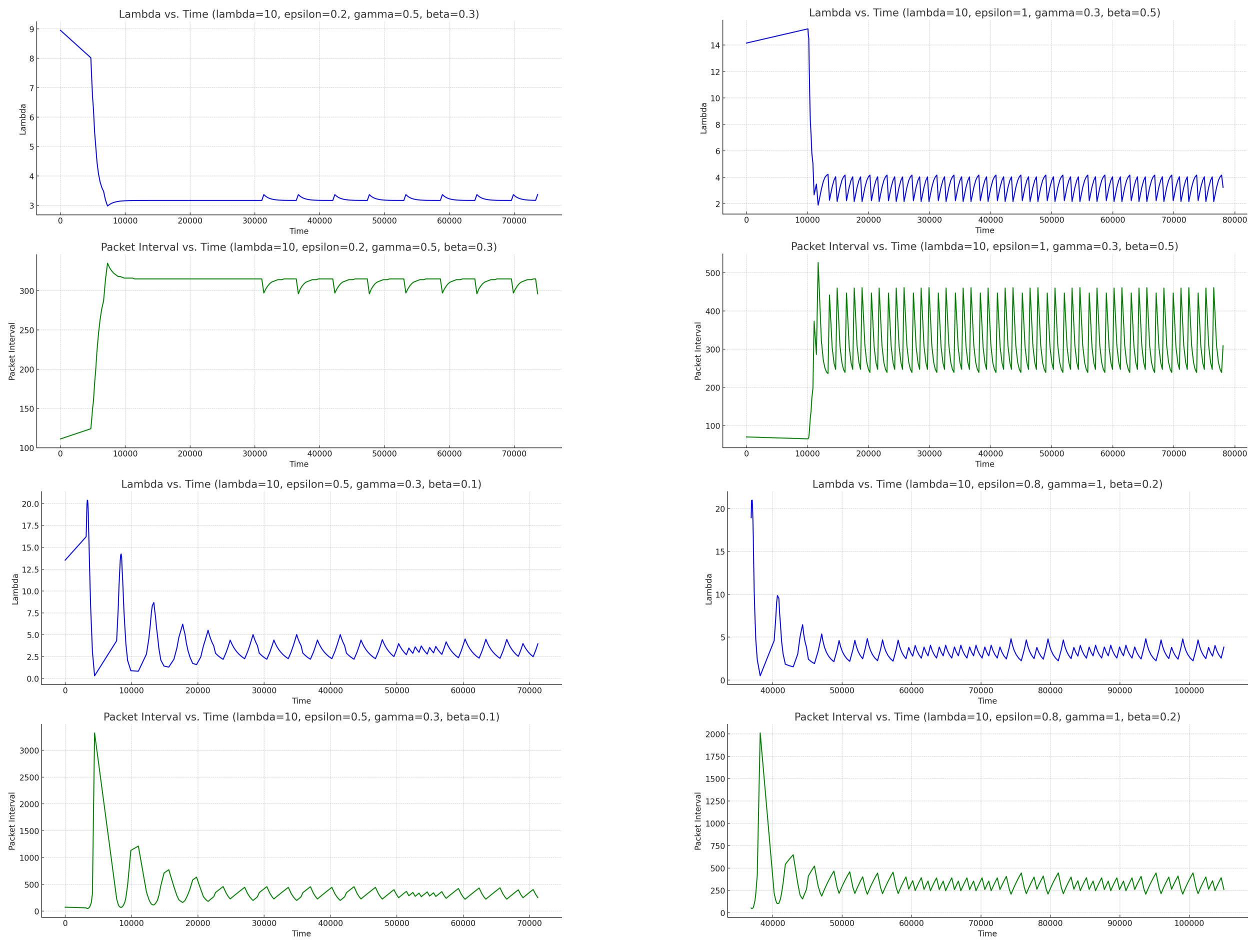


Figure 1: Lambda and TimeInterval change with method D

3.2 Buffersize

In this section we will compare the changes in the occupancy of the receiving buffer in the client side as seen in figure below.

- **Graph 1:** $\epsilon = 0.2$, $\beta = 0.3$
  - Initial buffer size reduction is less severe compared to the other graphs.
  - Shows a relatively stable pattern with minor fluctuations between 20,000 and 25,000.
  - The lowest ϵ value of all the graphs results in the least reactive changes, and a moderate β facilitates a stable equilibrium without significant oscillations.
- **Graph 2:** $\epsilon = 1$, $\beta = 0.5$
  - Initial buffer size drops sharply, then oscillates between approximately 28,000 and 32,000.
  - High ϵ causes strong reactions to changes, and a moderate β allows for a quick but not immediate stabilization.
- **Graph 3:** $\epsilon = 0.5$, $\beta = 0.1$
  - Buffer size shows large initial spikes with high variability, peaking at nearly 50,000.
  - The system exhibits a decrease in oscillation amplitude over time, stabilizing between 20,000 and 30,000.
  - The lower ϵ and β values lead to more dampened responses and a gradual approach to equilibrium.
- **Graph 4:** $\epsilon = 0.8$, $\beta = 0.2$
  - Starts with high volatility with buffer size peaks around 35,000.

- Decreases to a range between 20,000 and 30,000 with less pronounced oscillations compared to Graph 1, due to a lower ϵ.
- The lower β value leads to more gradual adjustments, resulting in a smoother trend after initial fluctuations.

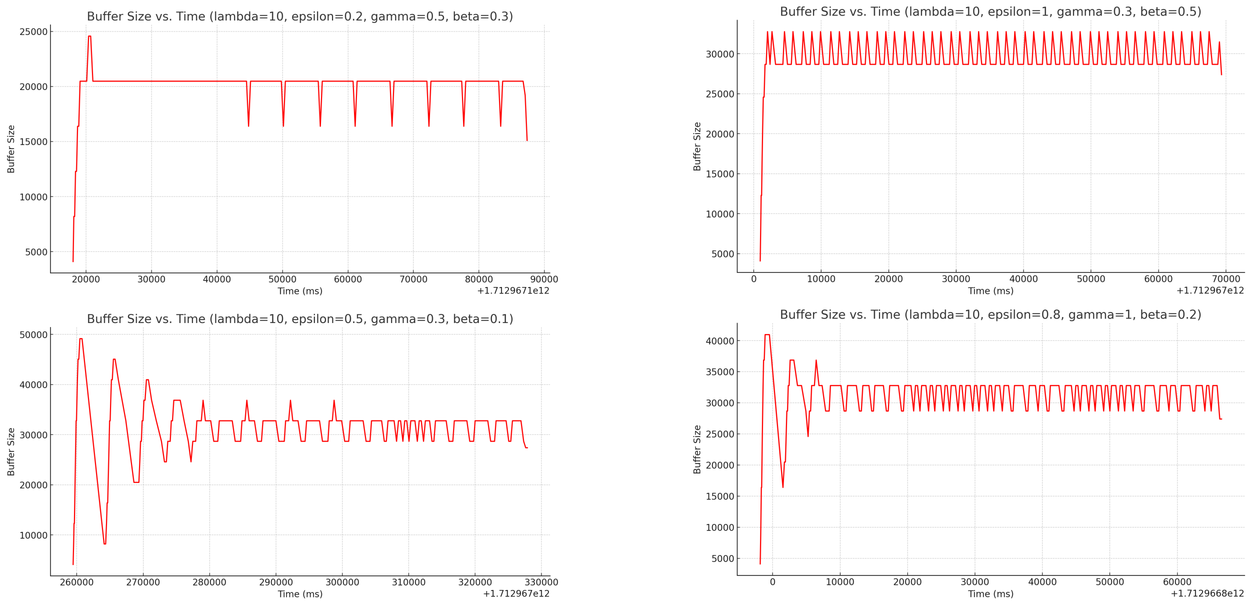


Figure 2: Buffersize change with method D

3.3 Method C

With Epsilon=0.5 : The buffer size fluctuates widely at the beginning, with sharp and frequent peaks and valleys. This suggests that the buffer experiences high variability in its occupancy, which might be indicative of a system that is more reactive to incoming traffic, possibly allowing more packets in before reacting to the congestion.

With Epsilon=0.1: The plot starts with a few sharp drops but quickly reaches a more stable pattern with less severe fluctuations than the first plot. This indicates that a lower epsilon results in a smoother buffer size over time, suggesting that the system might be more conservative or proactive in managing traffic, possibly by reducing the rate at which it accepts new packets earlier or more aggressively.

From these observations, we can infer that epsilon likely influences the aggressiveness of the buffer management policy. A higher epsilon leads to greater swings in buffer occupancy, implying a more lenient system that allows more variation before compensating. A lower epsilon appears to make the system more stable with fewer and less severe fluctuations, suggesting a tighter control on the buffer size.

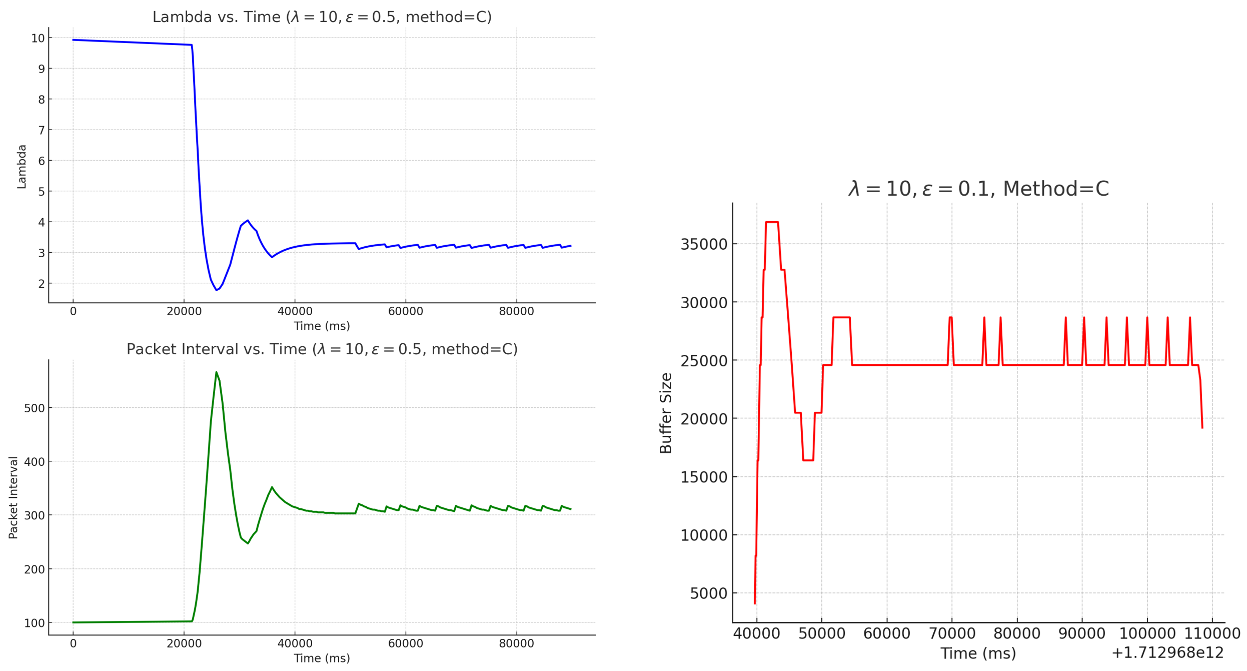
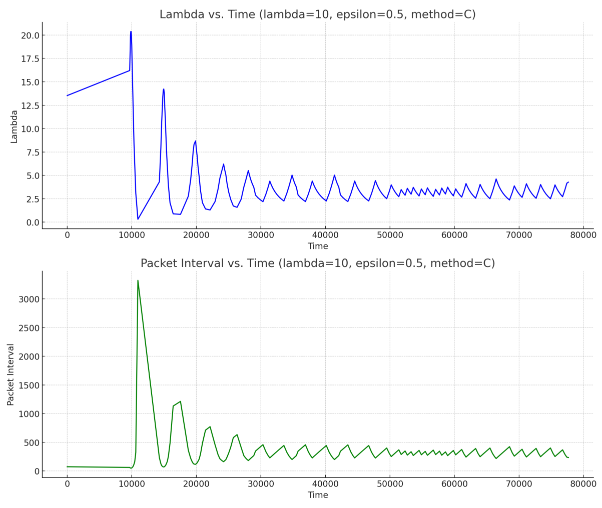


Figure 3: lambda and buffersize changes with epsilon 0.1



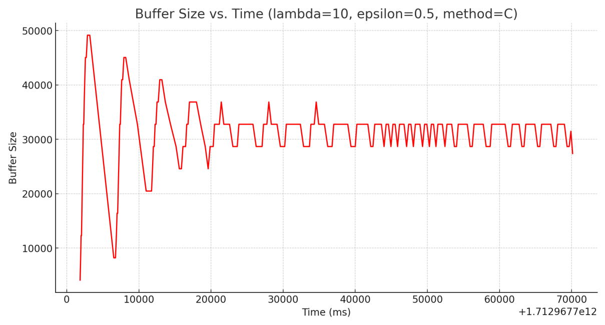


Figure 4: lambda and buffersize change with epsilon 0.5