Technical Report of Chorus (ACM MobiCom 2024)

The system architecture of Chorus is shown in Figure 1. The remainder of this document provides details on the Chorus implementation, which are omitted in the paper due to space limitations.

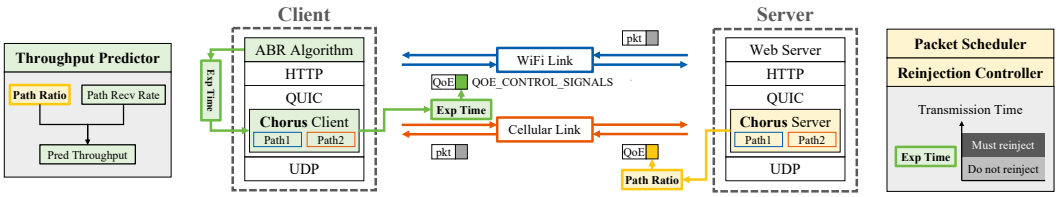

Fig. 1. Chorus system architecture.

1 QOE INTERACTION

Table 1. Fields in the QoE frame.

For	Field	Meaning
Common	qoe_seq_num	sequence number of QoE
	qoe_type	from the server (0x01) or the client (0x02)
	chunk_index	index of the chunk
Server	fast_path_idx	index of the fast path
	slow_path_idx	index of the slow path
	path_ratio	ratio of the fast path
Client	exp_time	expected time of the chunk

Chorus introduces the bidirectional QoE transmission for the two endpoints (the client and the server) to exchange information from different layers. Table 1 shows the QoE frame fields used by Chorus and their meanings. The QoE interaction acts as follows:

- On the client side, after the ABR algorithm determines the bitrate of the next chunk, the player calculates the corresponding predicted time and sends it to the server, contained in a QoE frame (qoe_type = 0x02), before the corresponding HTTP request.
- On the server side, the transport layer periodically (every 200ms) updates paths' statistics (i.e., bandwidth), and sends this information to the client via a QoE frame (qoe_type = 0x01).

In extreme scenarios, the client or the server may not receive the QoE frame from the other side. If this event happens, the player will predict throughput by HM (harmonic mean, see §3.3.2 in the paper), and the server will take the expected time as 0, i.e., conducting unlimited reinjection.

2 THE STATE MACHINE OF CD&FC

Chorus maintains an internal state machine at the server to divide the CD and FC phases, as shown in Figure 2. The state machine contains four states: *INIT*, *CD*, *FC_1*, and *FC_2*.

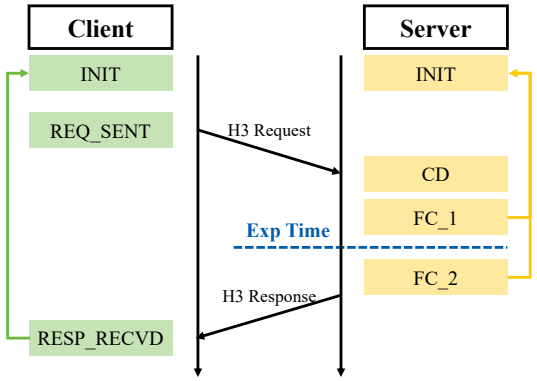

Fig. 2. State machine of CD&FC in Chorus.

- In the INIT state, Chorus uses MinRTT to transmit packets such as QoE frames. When the server application receives an HTTP/3 request, it will immediately inform the transport layer with the response (a chunk) starting through an added API (*update_client_resp_info*, see below). Then, the transport layer records the timestamp (T_s in Alg. 1 in the paper) and transits to the CD state.
- When receiving the response data of a chunk passed by the upper application layer, the transport layer at the server side conducts the one-shot scheduling for all packets, starts the transmission, and enters the FC_1 state, i.e., the 1st-stage correction.
- Chorus keeps examining the transmission time of the chunk. Once the transmission time is beyond expected (Eq. 10 in the paper), Chorus enters the FC_2 state with the 2nd-stage correction. After the chunk is transmitted, Chorus returns to the INIT state.

Note that the server may not know if the client has already received the last bytes of the chunk, especially when there are still unacknowledged reinjected or retransmitted packets on paths. In this case, Chorus will stay in the FC_1 or FC_2 state, wait for a new request to arrive, and then directly transit to the CD phase.

3 APIS AND CALLBACK FUNCTIONS

Chorus incorporates additional APIs and callback functions to facilitate interactions between the transport layer and the application layer. These APIs work in conjunction with the QoE frame fields defined in Table 1.

- *send_client_qoe_info(chunk_index, exp_time)*: An API for the client application (the video player) to send a QoE frame (qoe_type = 0x02) to the server. The player calls this API before sending an HTTP request, providing the chunk index and the expected delivery time calculated by the ABR algorithm.
- *update_client_resp_info(chunk_index, chunk_size)*: An API for the server application (the web server) to inform the transport layer when an HTTP response starts. The transport layer uses this information to record the chunk boundary and initiate the CD phase (see §2).
- *update_server_qoe_info(fast_path_idx, slow_path_idx, fast_path_bw, slow_path_bw, path_ratio)*: A callback function invoked when the client receives a QoE frame (qoe_type = 0x01) from the server. The client application uses the path statistics to compute throughput predictions for the ABR algorithm (see §4).

4 VIRTUAL PLAYER LOGIC

To conduct tests in emulation, we have implemented a virtual video player in XQUIC by modifying its test_server.c and test_client.c, containing the following logic:

- *Buffer management logic*: The playback buffer contains unplayed video frames. It decreases at the constant 1:1 speed (mimicking real-world video playback) and increases by the chunk duration (4 seconds) after downloading a chunk. Note that the buffer level stays at 0 when a rebuffering event occurs.
- *Chunk request logic*: The request pattern of a typical DASH video player (e.g., dash.js) exhibits the periodical ON-OFF pattern, which means it will not request a new chunk if the playback buffer is adequate. Referring to dash.js, we set a timer to check the buffer level periodically. If the buffer level is below the target value (30 seconds in emulation), the player will immediately send an HTTP/3 request; otherwise, it will wait for 0.5 seconds before the next check.

4.1 Chunk Download State Machine

The virtual player maintains a 3-state finite state machine to manage the chunk download lifecycle, as shown in Figure 3.

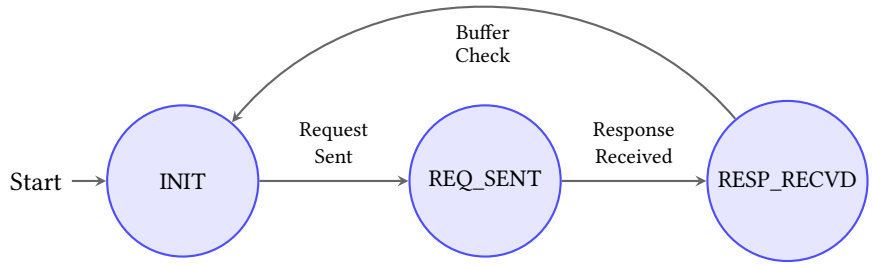

Fig. 3. State machine of chunk download in the virtual player.

- *INIT*: The player executes the ABR algorithm to select the bitrate for the next chunk. For Chorus, it also sends a QoE frame to the server before the HTTP request.
- *REQ_SENT*: The HTTP/3 request has been sent, waiting for the server response.
- *RESP_RECVD*: The chunk has been received. The player checks the buffer level and either requests the next chunk immediately (if buffer < TARGET_BUFFER) or waits before the next check.

4.2 ABR Algorithm: MPC

The virtual player uses Model Predictive Control (MPC) as the ABR algorithm, following the implementation in https://github.com/hongzimao/pensieve/blob/master/test/mpc.py. MPC selects the bitrate that maximizes the predicted QoE over a lookahead window:

- *Prediction window*: The player considers the next 5 chunks when making bitrate decisions.
- *Solution space*: With 5 bitrate levels and a 5-chunk window, MPC evaluates $5^5 = 3125$ possible bitrate combinations.
- *Objective function*: For each combination, MPC computes the predicted QoE using Eq. 1 in the paper:

$$\text{QoE} = \sum_{k}^{K} R_k - \mu \sum_{k}^{K} T_k - \lambda \sum_{k}^{K-1} |R_{k+1} - R_k| \tag{1}$$

where R_k is the bitrate of chunk k (Mbps), T_k is the rebuffering time (seconds), $\mu = 16$ is the rebuffering penalty (equal to the highest bitrate), and $\lambda = 1$ is the bitrate switching penalty.

4.3 Throughput Prediction

The accuracy of throughput prediction significantly affects ABR performance. The virtual player supports two prediction methods:

- *Harmonic Mean (HM)*: Computes the harmonic mean of the throughput samples from the past 5 chunks. This is a commonly used baseline method in ABR research.
- *Chorus Prediction*: When QoE frames from the server are available (see §1), the player uses path-aware throughput prediction. Given the receiving bandwidth of the fast path RB_f, the slow path RB_s, and the path ratio α (fraction of data on the fast path), the predicted throughput is:

$$\hat{C} = \min\left(\frac{\hat{RB}_f}{\alpha}, \frac{\hat{RB}_s}{1-\alpha}\right) \tag{2}$$

  This formula accounts for the multipath scheduling decision and provides more accurate predictions than single-path methods. In practice, Chorus further limits the predicted throughput as $\hat{C}_k \geq \max\{\hat{RB}_f, \hat{RB}_s\}$, corresponding to the footnote #2 in §3.3.2 of the Chorus paper.

4.4 Key Parameters

Table 2 summarizes the key parameters used in the virtual player implementation.

Table 2. Key parameters in the virtual player.

Parameter	Value	Description
TARGET_BUFFER	30 s	Target playback buffer level
CHUNK_DURATION	4 s	Duration of each video chunk
BIT_RATE	{1, 2.5, 5, 8, 16} Mbps	Available bitrate levels
MPC_HORIZON	5 chunks	Lookahead window for MPC
HM_SAMPLES	5	Sample count for harmonic mean
μ	16	Rebuffering penalty coefficient
λ	1	Bitrate switching penalty

5 EMULATION TEST CONFIGURATION

The emulation experiments use configuration files to define test scenarios. The experimental data and network traces are available at https://github.com/GreenLv/Chorus.

5.1 Test Configuration File Format

Each algorithm directory contains a `.test` file that specifies the test configuration. The format is as follows:

```
Line 1: {algorithm} {duration_seconds}
Line 2: {path_type} {cc_count} {congestion_control}
Line 3+: {trace1} {owd1} {loss1} {size1} {trace2} {owd2} {loss2} {size2}
```

Table 3 describes each field in the configuration file.

Table 3. Fields in the test configuration file.

Field	Description
algorithm	Algorithm name (Chorus, XLINK, MinRTT, MinRTTRI, SP)
duration_seconds	Test duration in seconds
path_type	MP (multipath) or SP (single-path)
cc_count	Number of congestion control algorithms
congestion_control	Congestion control algorithm (e.g., Cubic)
trace1, trace2	Network trace names for each path
owd1, owd2	One-way delay in milliseconds for each path
loss1, loss2	Packet loss rate for each path
size1, size2	Link buffer size in bytes

5.2 Test Naming Convention

Each test directory under `tests/` follows the naming format:

`{test_number}~{trace_name_path1}~{owd_path1}~{trace_name_path2}~{owd_path2}`

For example, `1~cellular_subway_3~25~cellular_airport~35` represents Test #1 using the `cellular_subway_3` trace with 25ms one-way delay on Path 1, and the `cellular_airport` trace with 35ms one-way delay on Path 2.