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.

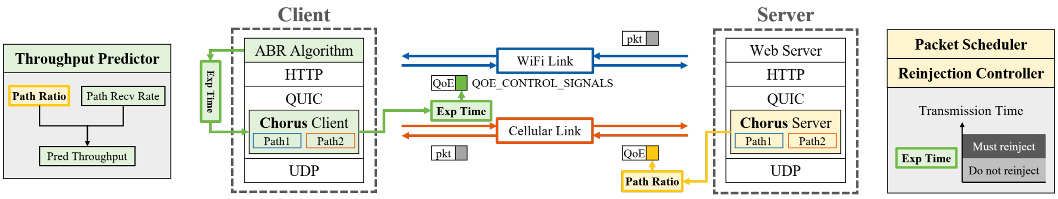

Fig. 1. Chorus system architecture.

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 Local 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*.

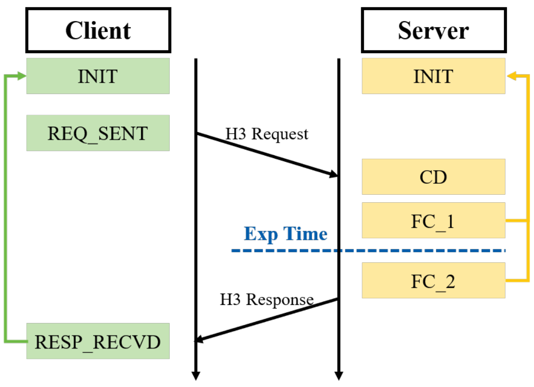

Fig. 2. State machine of CD&FC in Chorus.

- 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 already receives 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 with the application layer, including:

- *send_client_qoe_info*: an API for the client application (the video player) to send qoe frame, containing the expected time of each chunk.
- *update_client_resp_info*: an API for the server application (the web server) to inform the transport layer of the start time and the chunk size corresponding to each HTTP request. This API is important for Chorus to be aware of the chunk boundary.
- *update_server_qoe_info*: a callback function for the client application to inform its transport layer to update information in the QoE frames sent by the server.

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.