CGReplay: Capture and Replay of Cloud Gaming Traffic for QoE/QoS Assessment

Alireza Shirmarz, Ariel G. de Castro, Fabio L. Verdi, Christian E. Rothenberg
Universidade Federal de São Carlos (UFSCar) - SP, Brazil
{ashirmarz, verdi}@ufscar.br
Universidade Estadual de Campinas (UNICAMP) - SP, Brazil
a272319@dac.unicamp.br chesteve@dca.fee.unicamp.br

***Abstract*—Cloud Gaming (CG) research faces challenges due to the unpredictability of game engines and restricted access to commercial platforms and their logs. This creates major obstacles to conducting fair experimentation and evaluation. `CGReplay` captures and replays player commands and the corresponding video frames in an ordered and synchronized action-reaction loop, ensuring reproducibility. It enables Quality of Experience/Service (QoE/QoS) assessment under varying network conditions and serves as a foundation for broader CG research. The code is publicly available for further development[1].**

***Index Terms*—Cloud Gaming, QoE, QoS, Interactivity, CG Traffic Generator.**

I. Introduction & Motivation

Cloud Gaming (CG) is rapidly growing as a popular entertainment medium, making both objective and subjective Quality of Experience (QoE) critical areas of research and industry focus [1], [2]. However, most mainstream CG platforms (*e.g.*, Xbox Cloud Gaming, GeForce Now) are closed-source, hampering controlled research experiments and evaluation, restricting the exploration of innovative research directions in areas such as frame generation techniques, novel QoE metrics development, latency optimization strategies, and adaptive streaming algorithms tailored specifically for interactive gaming contexts. Additionally, the inherently non-deterministic nature of gameplay means that the same scene is never repeated, preventing direct comparison of video frames across different network conditions. Since a CG session involves uplink commands (player input) and downlink video frames (server response), their interactive behavior – particularly, response time – is essential to QoE. Therefore, a platform that can capture, synchronize, and replay these interactions is essential for reproducible testing and QoE assessment.

While open-source CG platforms such as GamingAnywhere[2] and Moonlight Game Streaming[3] exist, they also present challenges for QoE evaluation due to gameplay non-determinism – where commands and video frames vary unpredictably. This variability makes it difficult to assess downlink video frame quality, uplink command accuracy, and their interactive influences under different network conditions. For generating CG traffic and evaluating its QoE, a platform is needed to capture CG over the Internet and replay the same uplink and downlink sequences. Such a solution would allow researchers to compare received video frames and evaluate how network conditions affect interactivity between commands and frames.

Experimental Cloud Gaming Platform (eCGP) [2] aimed to achieve deterministic cloud gaming outputs by bypassing the non-determinism of game engines for Quality of Experience (QoS) evaluation. However, this approach does not fully support the automatic interactivity between uplink and downlink. Moreover, they fail to address real-world cloud gaming challenges, such as command and frame loss, which impact fidelity. Inspired by TCPReplay, we propose `CGReplay`[4], an open-source, configurable platform that captures and replays CG sessions at the application layer while preserving automatic interactivity. By synchronizing the timing and sequence of commands and frames, `CGReplay` allows researchers to conduct reproducible QoE experiments under various network conditions, ensuring higher fidelity to real-world cloud gaming environments. `CGReplay` is organized into two main phases:

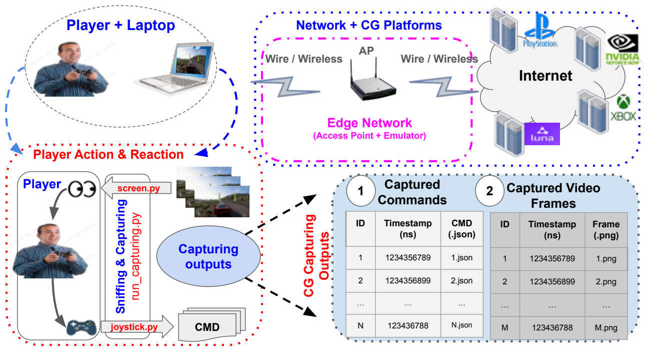
1) **CG Capturing:** in this phase, the platform records uplink commands and downlink video frames along with their interaction, ensuring that the complete behavioral pattern of a cloud gaming session is preserved.
2) **CG Replaying:** the recorded data is synchronized and replayed to mimic the original session, enabling reproducible testing under diverse network conditions.

For this demonstration, our implementation uses UDP, RTP, and SCReAM[5], however, as an open-source platform, it can be extended for other streaming protocols such as RTP over QUIC (RoQ) [3]. `CGReplay` can also be used as the platform for evaluation end development of other research CG adaptive rendering improvement [1], the adaptive encoding [4] for CG and Generative interactive environments (Genie) [5].

II. Highlights of CGReplay

`CGReplay` is an open-source platform for capturing and replaying bi-directional cloud gaming sessions (Ac-

[1]https://github.com/dcomp-leris/CGReplay.git.
[2]https://github.com/chunying/gaminganywhere.git.
[3]https://github.com/moonlight-stream/moonlight-stream.github.io.git.
[4]This is a complementary work supporting the full paper '*In-Network AR/CG Traffic Classification Entirely Deployed in the Programmable Data Plane*' accepted at Netsoft 2025 main track.
[5]https://github.com/EricssonResearch/scream.git.



ID	Timestamp (ns)	CMD (.json)
1	1234356789	1.json
2	1234356899	2.json
...	...	...
N	123436788	N.json

ID	Timestamp (ns)	Frame (.png)
1	1234356789	1.png
2	1234356899	2.png
...	...	...
M	123436788	M.png

Fig. 1. Capturing online CG. The AP is connected to the Internet via a wired connection and connects to the laptop using Wi-Fi 6.

tion/Reaction) based on real online CG interactions. It offers easy configuration via YAML files for each component, ensuring flexibility and simplicity. Its main contributions are structured into three key parts: capturing, replaying, and ensuring action/reaction interactivity and reliability. In the capturing phase, the platform records uplink commands and downlink video frames while extracting their sequential patterns. During replaying, these sequences are utilized by two agents: (a) *CG server* and (b) *CG player* running on Python-supported systems, to faithfully mimic the original session. Finally, the action/reaction interactivity module emulates user actions and server responses, with SCReAM (optional) integration for UDP-based congestion control to evaluate ECN-based signaling alongside traditional drop mechanisms.

A. Capturing the Cloud Gaming

In this phase, the module *run_capturing.py* captures both uplink commands (sniffed via the USB port) and downlink video frames (captured from the screen) at a configurable sampling rate of 30 frames per second. In our demonstration, a human player uses an Xbox cloud gaming platform to play three games – Forza Horizon 5, Fortnite, and Mortal Kombat 11, as shown in Fig. 1. This flexible setup accommodates various platforms and topologies (*e.g.*, PlayStation, GeForce Now, Luna) and generates two synchronized files: one containing commands stored in a JSON file and the other containing video frames as PNG files, both annotated with IDs and timestamps. These logs are then used to extract a synchronized command/frame sequence for accurate sequential command and frame replay. Although the capturing module can record various inputs, in this study we focus on the joystick – specifically, the Xbox Controller.

B. Replaying the Cloud Gaming

CGReplay comprises two Python agents – (a) *CG server* and (b) *CG player* – deployed on separate hosts that can be interconnected via physical devices or simulated/emulated networks (see Fig. 2). These modules use IPv4 and UDP for both uplink and downlink communications. Video streaming is

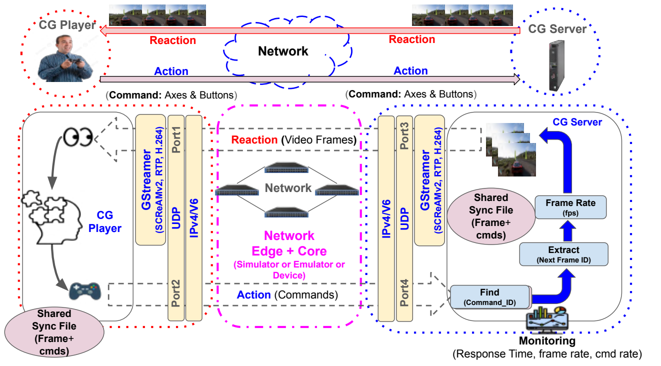
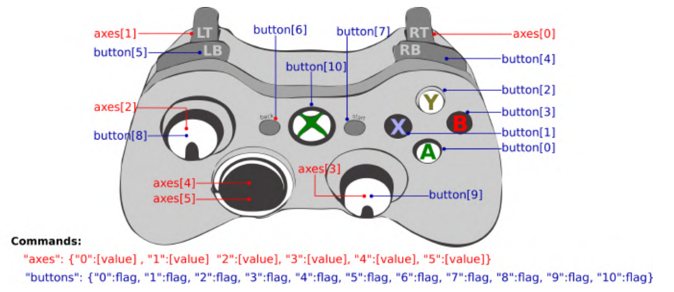

CG Player
Reaction
Network
Reaction
CG Server
Action
Action
(Command: Axes & Buttons)
(Command: Axes & Buttons)
CG Player
GStreamer (SCReAMv2, RTP, H.264)
UDP
IPv4/V6
Port1
Port2
Reaction (Video Frames)
Network
Network Edge + Core
(Simulator or Emulator or Device)
Action (Commands)
IPv4/V6
UDP
GStreamer (SCReAMv2, RTP, H.264)
Port3
Port4
CG Server
Shared Sync File (Frame+ cmds)
Frame Rate (fps)
Extract (Next Frame ID)
Find (Command_ID)
Shared Sync File (Frame+ cmds)
Monitoring
(Response Time, frame rate, cmd rate)

Fig. 2. Replaying CG traffic with CGReplay agents at server and player sides.

implemented with GStreamer[6], with RTP and H.264 encoding, with optional SCReAM integration as a scalable congestion control mechanism. The synchronized sequence of commands and frames ensures sequential replay between the *CG player* and *CG server*. During replay, the CG server streams frames sequentially until a command is needed. The CG player processes each frame, triggers the corresponding command, and sends it to the server. Streaming resumes upon receiving the expected command, ensuring proper interactivity.

Game Video Frame: video frames are stored as PNG files in the CG server's folder and sent in response to specific commands. Each frame is dynamically labeled with a unique ID and timestamp embedded as a QR code, allowing the CG player to verify frames during replay.

Commands: commands are stored in JSON format with "axes" for joystick movements and "button" for presses (see Fig. 3). Each command includes a unique ID, timestamp, and ACK/NACK, ensuring synchronization and sequence order between the CG server and player.



```
Commands:
  "axes": {"0":[value] , "1":[value]  "2":[value], "3":[value], "4":[value], "5":[value]}
  "buttons": {"0":flag, "1":flag, "2":flag, "3":flag, "4":flag, "5":flag, "6":flag, "7":flag, "8":flag, "9":flag, "10":flag}
```

Fig. 3. Xbox Joystick Button Mapping with Corresponding Commands. The axes' continuous values range [-32767, 32767], and the button flag is 0 or 1.

C. Action/Reaction Interactivity & Reliability Mechanism

In a real-time CG platform, the interactive loop works by having the player watch the current video frames, make decisions (*e.g.*, pressing buttons), and then see updated frames influenced by those commands. To replicate this behavior automatically in CGReplay, the system uses a shared "sync order" of frames and commands, each with unique IDs (F_{ID} for

[6]https://gstreamer.freedesktop.org/.

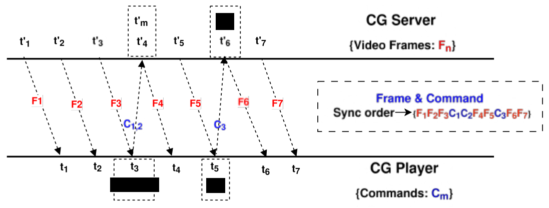


Fig. 4. Action & Reaction Interactivity in CGReplay.

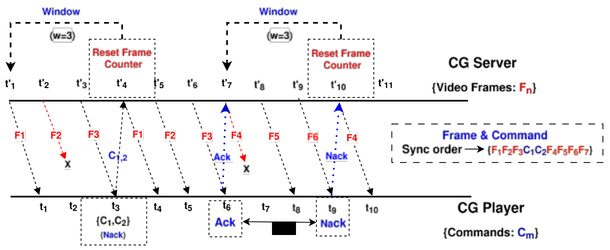


Fig. 6. Frame f_2 is lost while f_3 is received. The player notifies with NACK. ACK is sent periodically ($w = 3$). NACK is sent at t_{10} for frame f_4 loss.

frames, C_{ID} for commands). As shown in Fig. 4, an example sync order might be $\{F_1, F_2, F_3, C_1, C_2, F_4, F_5, C_3, F_6, F_7\}$. Both the *CG server* and the *CG player* compare incoming IDs against this sequence to ensure each action (command) is followed by its corresponding reaction (frame) at the correct time. Meanwhile, on the *CG server* side, frames are streamed sequentially, but the server pauses at specific times t'_m to wait for new commands before rendering the next frame, as illustrated in Fig. 4. For instance, at t'_4 the server expects commands $\{C_1, C_2\}$, and at t'_6 it waits for C_3. This enforced timing ensures that every newly rendered frame accurately reflects the latest player input, maintaining the realistic cycle of action and reaction found in a real-world CG platform.

`CGReplay` interactivity faces two challenges: *command loss* and *frame loss*, requiring reliability mechanisms.

Command Loss Mechanism: command loss in CGReplay can occur due to network conditions. To address this, a recovery mechanism is implemented in both the server and player. As shown in Fig. 5, if $\{C_1\ ,C_2\}$ is lost, the CG server, while waiting for them, retransmits the previous frame to signal the player of the missing command – or due to excessive delay. Upon detecting two consecutive frames with the same F_{ID}, the player re-sends the previous command. The CG server resumes sequential frame streaming once it receives the expected commands, ensuring continuity in interactivity.

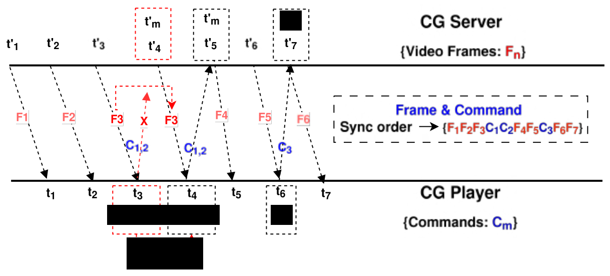


Fig. 5. Command loss scenario. $\{C_1, C_2\}$ are lost through the network. Frame f_3 is retransmitted and the CG server waits for $\{C_1, C_2\}$.

Frame Loss Mechanism: to handle frame loss, we implement an ACK/NACK control mechanism to keep the CG player synchronized with the server. ACKs and NACKs are sent in two ways: (1) as metadata with commands and (2) periodically after receiving a defined window w of frames. When the player receives w frames, it checks their F_{ID}s against the expected sync order – sending an ACK if correct or a NACK if frames are missing or out of order. Upon receiving an ACK, the CG server continues streaming sequentially. However, if a NACK is received, the server rolls back w frames and retransmits from that point. As shown in Fig. 6, a NACK sent with C_1, C_2 prompts the server to backtrack from the next expected Frame ID ($F_{ID} = 7$) to $F_{ID} = 4$, calculated as (7 - window size of 3). The server then resumes streaming from $F_{ID} = 4$, ensuring reliable frame delivery. To mitigate the impact of high latency, we introduce window sliding in the CG server, allowing it to continue streaming up to w frames instead of pausing at the expected frame. This increases tolerance to delays and prevents excessive interruptions, making the `CGReplay` platform smoother and more responsive.

III. Conclusion, Demo & Future Work

In this work, we present `CGReplay`, an open-source platform that captures and replays user gaming sessions for CG. `CGReplay` is fully configurable through YAML files for both server and player agents and provides detailed outputs, including received video frames, frame rate, command rate, command/frame loss reports, and action/reaction response time. These features enable CG QoE evaluation under different network conditions. Additionally, `CGReplay` lays the foundation for future advancements in intelligent CG platforms and optimized encoding for CG.

During the demo: Participants can replay different games on a single computer using a client-server setup in Mininet. They can view video frames, commands, and QoS/QoE metrics like FPS. Settings such as SCReAM and target FPS are adjustable. Mininet is used for simplicity but can be replaced by realistic platforms such as Tofino switches or a more complex topology.

Acknowledgment

This work was supported by Ericsson Telecomunicações Ltda., and by the Sao Paulo Research Foundation (FAPESP), FAPESP grant `2021/00199-8`, CPE SMARTNESS.

References

[1] J. Heo *et al.*, "Adrenaline: Adaptive rendering optimization system for scalable cloud gaming," *arXiv preprint arXiv:2412.19446*, 2024.
[2] P. Graff *et al.*, "Improving cloud gaming traffic qos: a comparison between class-based queuing policy and l4s," in *8th Network Traffic Measurement and Analysis Conference (TMA) (IEEE)*, 2024.
[3] M. Engelbart *et al.*, "RTP over QUIC (RoQ)," Internet-Draft draft-ietf-avtcore-rtp-over-quic-13, Internet Engineering Task Force, Feb. 2025. Work in Progress.
[4] S. Fouladi *et al.*, "Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol," in *15th USENIX NSDI*, pp. 267–282, 2018.
[5] J. Bruce *et al.*, "Genie: Generative interactive environments," in *ICML*, 2024.