Talaria: In-engine Synchronisation for Seamless Migration of Mobile Edge Gaming Instances

Tristan Braud
The Hong Kong University of Science and Technology
Hong Kong

Ahmad Alhilal
The Hong Kong University of Science and Technology
Hong Kong

Pan Hui
The Hong Kong University of Science and Technology
Hong Kong
University of Helsinki
Finland

ABSTRACT

Mobile cloud gaming requires a very low end-to-end latency. Edge computing significantly reduces network latency. However, in mobility scenarios, the user will frequently move out of the edge server's coverage area, requiring frequent migration of the game instance. This paper presents Talaria, an in-engine content synchronisation solution for unnoticeable game instance migration between edge servers. Talaria creates a minimal instance with content immediately relevant to the game experience, allowing the client to switch servers in a minimal amount of time. The remaining content is then synchronised according to priority until the game's state is coherent between both instances. Our implementation of Talaria as a Unity engine plugin reduces the game's downtime by 61% compared to one-off server migration, with an average latency below 25 ms for the server migration, and 87 ms for the entire game synchronisation.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Networks** → **Network mobility**; • **Software and its engineering** → *Interactive games*.

KEYWORDS

Cloud Gaming, Live Migration, Mobile Edge Computing

ACM Reference Format:
Tristan Braud, Ahmad Alhilal, and Pan Hui. 2021. Talaria: In-engine Synchronisation for Seamless Migration of Mobile Edge Gaming Instances. In *Proceedings of CoNEXT '21*. ACM, New York, NY, USA, 7 pages. https://doi.org/TBA

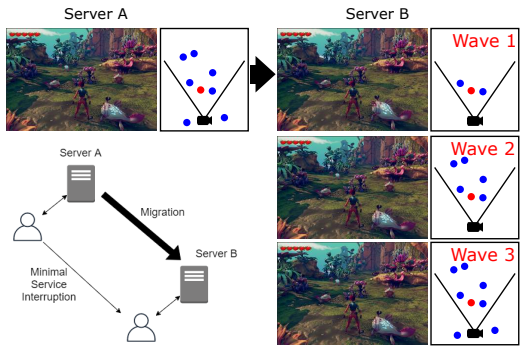


Figure 1: Talaria provides ultrafast migration of mobile edge gaming instances by iterative game element synchronisation waves: (1) player, camera, and nearby objects, (2) visible objects, (3) remaining objects.

1 INTRODUCTION

In recent years, mobile games have become increasingly resource-intensive. Games such as Genshin Impact[1], Wild Rift[2], or Player Unknown Battleground[3] rely on sophisticated 3D graphics and have millions to dozens of millions of users worldwide. Mobile cloud gaming enables users to experience such high-end games on constrained mobile devices by offloading computation-heavy operations to powerful remote machines. Gaming is highly interactive and requires very low end-to-end latency to ensure a satisfying user experience [3, 9, 24]. The edge computing paradigm significantly reduces network latency by bringing the computation resources closer to the user, ideally a single hop away. In a mobile setting, users frequently move out of the coverage area of the edge server, requiring frequent migrations of the application to maintain minimal latency. As a result, most edge applications are either stateless or rely on process checkpointing to migrate containers in a matter of seconds [6, 13].

Live game migration for mobile edge gaming faces significant challenges. Gaming relies on Graphics Processing Units (GPU) for rendering sophisticated 3D scenes and objects. This GPU usage is stateful, with objects and textures remaining in the memory between frames. As such, migrating gaming instances requires migrating the content of the GPU memory. GPU checkpointing has yet to be addressed for convenient live migration of game instances. Current solutions targeting GPU checkpointing for gaming and 3D rendering take dozens of seconds to checkpoint and restore an instance, notwithstanding the network transmission that may account for several GB of data [8, 18]. Migration thus leads to a significant pause in the gaming experience. Finally, 5G aims to densify the edge computing infrastructure, leading to users frequently moving between edge server coverage areas and thus frequent migration for stateful

[1] https://genshin.mihoyo.com/
[2] https://wildrift.leagueoflegends.com/
[3] https://www.pubgmobile.com/

CoNEXT '21, December 7–10, 2021, Virtual Event, Germany

ACM ISBN TBA. . . $TBA
https://doi.org/TBA

applications such as games. Under such conditions, frequent and lengthy migrations will considerably degrade the user experience.

To address these issues, we develop Talaria, an in-engine solution composed of a game migration protocol and an object-based migration algorithm for seamless video game instance migration. By operating in-engine, Talaria only synchronises the elements that are critical to the game state's coherence, significantly outperforming checkpoint-based solutions that recreate perfect clones of the instance by dumping the entire memory content. Talaria's migration leads to service interruptions below 25 ms, with 44 ms between the two frames during which the migration happens. Coincidentally, operating in-engine also reduces the number of unnecessary memory copies, leading to faster frame capture, encoding, and transmission. To achieve such low migration time, Talaria proactively starts a new game instance on the next server. When the migration starts, Talaria saves the game objects' state and synchronises them between both game instances in several waves. As shown in Figure 1, the objects are synchronised according to their importance for the user experience, starting with the player, camera, and nearby objects (wave 1), and ending with objects that are not visible by the user (wave 3). Talaria synchronises the visible objects down to the exact animation frame, enabling a seamless transition while significantly reducing the network load compared to checkpointing.

Our contribution can be summarised as follow:

(1) **Design** of Talaria, an in-engine framework allowing seamless migration of cloud and edge gaming instances.
(2) **Implementation** of Talaria as a Unity engine plugin and **integration** of Talaria within a sophisticated game project.
(3) **Evaluation** of Talaria: service interruption shorter than 25 ms, and total migration below 87 ms, while resulting in a network load below 30 KB; much faster than traditional checkpointing.

The rest of this paper is organised as follows: after reviewing the most relevant related works in Section 2, we describe Talaria's system architecture in Section 3 and detail its implementation in Section 4. We finally evaluate Talaria's performance in Section 4 and address the implications for mobile edge gaming in Section 6.

2 RELATED WORKS

Many works address live container migrations [23] but few consider GPU applications, and most do not offer acceptable performance for cloud gaming. This section first discusses container checkpointing before moving to GPU-bound migration. Finally, we discuss in-engine cloud gaming works that relate to Talaria's architecture.

Container checkpointing: Fast container migration has become a priority with multi-access edge cloud (MEC). Most modern solutions rely on container checkpointing. However, checkpointing can be costly if the application's memory footprint is high, leading to dozens to thousands of seconds of service interruption [14]. Many works aim to reduce the migration time through iterative pre-copying [12], intermediary Network File Storage (NFS) servers [1], or a pipeline of processing stages [7, 14]. However, these works do not target GPU applications such as gaming, evaluation is often performed on lightweight applications, and the systems migrate services in dozens of seconds, incompatible with seamless cloud gaming migration. In this paper, we significantly improve the migration performance by performing layered migration at the engine level and synchronising the game elements in order of importance.

GPU application migration: Migrating GPU applications has long been a significant problem, as GPU architectures prevent convenient checkpointing. Transparent Gaming [2] can provide gaming applications on top of any GPU-equipped mobile computing nodes. However, It does not consider seamless migration between nodes during a live gaming session. Most works focus on CUDA [5] or OpenCL [17] applications [21, 25], which rely on a specific set of instructions. Among these studies, Prades et al. [19] use GPU virtualisation for checkpointing, which could be expanded to OpenGL applications. Few studies address replaying OpenGL calls [8, 11, 18]. Although the technique is promising, the performance is still too limited (i.e., 4 seconds to restart Maya from a checkpoint image [8]) for real-time cloud gaming migration. Besides, not all solutions consider GPU-accelerated OpenGL checkpointing [8]. In this paper, we circumvent this issue by bootstrapping the game instance and iteratively synchronising the game state, leading to significantly shorter migrations and lower network load.

In-engine cloud gaming: Although this paper is the first study on in-engine edge cloud gaming migration, several works considered in-engine cloud gaming for improving performance. Kahawai [4] renders a low-quality version of the game frames at the client, which is combined with the high-quality frames returned by the server to reduce the network load and enhance reliability. Similarly, Messaoudi et al. [16] devise a framework where only part of the game is rendered on the server. Finally, Outatime [10] renders multiple frames corresponding to several predicted outcomes and lets the client pick the appropriate frame. These works demonstrate the feasibility of in-engine cloud gaming and pave the way for in-engine mobile edge gaming migration such as Talaria.

3 SYSTEM ARCHITECTURE

We consider the following scenario: a user plays a game executed on an edge server (server A) while commuting. The lightweight client transmits the user controls to the server, which renders the game frames[4], encodes them, and transmits them over the network. The client then decodes and buffers the frames for display. Rendering frames is a GPU-intensive stateful operation that cannot be easily migrated through checkpointing. During a gaming session, the user moves to a location within the range of another server (server B), going through a migration area covered by both servers. Figure 2 summarises the messages exchanged during the migration.

3.1 Proactive Game Instance Provision

When the user enters the migration area, Talaria's client sends a REQUEST message to server B to start a new game instance. Talaria only synchronises the static elements that are essential to the game session, such as the current game level or the player's general location in the game world. The game instance on server B then establishes a connection with server A. Preloading the instance on server B hides up to several minutes of loading times by requiring only fine-grained synchronisation when the actual migration occurs. At this stage, the client has a direct connection to both servers.

[4] In the rest of this paper, the term 'frame' refers to video frames exclusively.

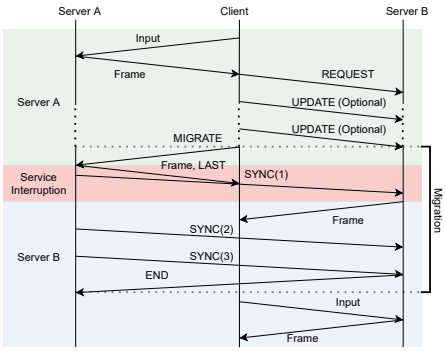


Figure 2: Primary messages exchanged during Talaria's migration. Talaria results in a minimal service interruption, with the entire downtime being in the order of a few round-trip times.

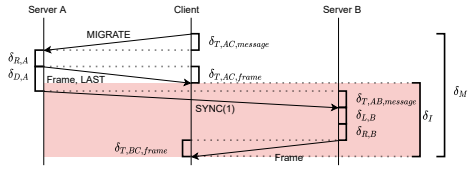


Figure 3: Time diagram of Talaria's service interruption during migration. The service interruption is unnoticeable if $\delta_I = \delta_{D,A} + \delta_{L,B} + \delta_{R,B} < 33\,ms$.

The preloaded game instance idles on server B until the user leaves the migration area. If the user leaves the migration area for server B's coverage area, migration occurs, and the preloaded instance becomes the active instance. On the other hand, if the user returns to the exclusive range of server A, the preloaded instance is destroyed, and the instance running on server A remains the primary instance. If the user stays in the migration area longer than expected, it can send additional messages to server B to update significant events (UPDATE). These messages help signify important game progress, for instance, the need to load a new level.

To minimise the resource usage of the preloaded instance, it is critical to predict the user's motion accurately. Most mobile game players game while commuting, going through well-established paths, which considerably simplifies the prediction task. Another solution could be decorrelating network handover and instance migration by initiating the prewarm instance only once the user has moved into the server's coverage area.

3.2 Migration

When the user leaves the migration area for server B's coverage area, it triggers the migration by sending a message to server A (MIGRATE). Upon receiving the MIGRATE message, server A transmits the last frame to the client with a specific tag (LAST) and starts the migration with server B. The client then closes the connection with server A and exclusively uses the connection with server B for sending player input and receiving the game frames.

Immediately after sending the last frame, server A starts the migration process. The state of the objects is dumped on server A and synchronised with server B (SYNC messages). Talaria aggregates the objects in three groups corresponding to three migration waves to accelerate the migration process and minimise the downtime for the client, with the first wave being sufficient to recover the immediate action's state on server B seamlessly. In practice, the pause duration can be made effectively unnoticeable to the user, contrary to checkpointing-based solutions (see Section 3.3)

The first migration phase encompasses all essential objects for the current game state. Upon receiving the first SYNC message, server B immediately synchronises the essential components and transmits a frame to the client. We expect inter-server connections to display a lower latency than the wireless client-server connection. As such, the entire network communication between the first MIGRATE message to Server A and the first frame from server B takes less than two round trip times (RTT). As Talaria transitions between game instances in a minimal time (ideally lower than the interframe time), it is necessary to preserve all the components involved in the current action, for instance, the main camera, the player character, and the objects near the player character. Talaria synchronises these components down to the exact frame of the animation to ensure the continuity between the frames sent by server A and server B. Server A then proceeds to the subsequent two migration phases, corresponding to the visible and remaining objects, respectively. Visible objects are also synchronised down to the state of the animation, while other objects only require their general state to be updated (transform, behaviour state, hit points). Although we made the choice in this paper to consider visible objects for the second wave, other strategies could be applied, such as synchronising all objects in a certain perimeter around the player character or only the objects that are engaged with the player character in their behaviour state machine. Once server B has updated the state of all objects, it signals the successful migration to server A (END message), which closes the connection with server B and terminates the instance.

3.3 Service Interruption

As shown in Figure 2, Talaria results in a minimal service interruption. The network transmission of the control messages spans over less than two round-trip times, assuming inter-server latency is lower than client-server latency. Between the reception of the MIGRATE message by server A and the generation of the first frame by server B, no content is generated or transmitted to the client. If the time between the reception of the last frame from server A and the first frame from server B δ_I is higher than the interframe time, the user will notice the service interruption. Let δ_M be the total migration time (from client sending MIGRATE to client receiving server B's first frame), δ_{SI} the service interruption time (time during which neither server A or server B work towards rendering or transmitting a frame), $\delta_{D,A}$ the time to dump the first wave of objects on server A, $\delta_{L,B}$ the time to load the first wave of objects on server B, $\delta_{R,A}$ and $\delta_{R,B}$ the rendering times on server A and B, respectively, and $\delta_{T,AB}$, $\delta_{T,AC}$, $\delta_{T,BC}$, the transmission time between server A and server B, server A and client, and server B and client, respectively. We consider eventual frame encoding and decoding time as part of their transmission time $\delta_{T,frame}$. We summarize these notations in Figure 3. Assuming that $\delta_{T,AC} < \delta_M$, we have:

$$\delta_M = \delta_I + \delta_{T,AC,mess} + \delta_{R,A} + \delta_{T,AC,frame} \quad (1)$$

$$\delta_I = \delta_{SI} + \delta_{T,BC,frame} - \delta_{T,AC,frame} \quad (2)$$

$$\delta_{SI} = \delta_{D,A} + \delta_{T,AC,frame} + \delta_{LB} \quad (3)$$

As such, the service interruption time will be unnoticeable to the user if δ_I is lower than the interframe time. At 30 FPS, we should have $\delta_I < 33\,ms$. If we consider servers A and B to be located on the same network and connected with very high speed wired links, and that the time to encode and transmit a game frame to the client is similar from server A and server B, we can approximate that $\delta_I = \delta_{D,A} + \delta_{L,B} + \delta_{R,B} < 33\,ms$. In practice, most cloud gaming systems buffer frames to account for unexpected latency variations. Given a buffer accounting for $\delta_t = 100\,ms$ latency (about 3 frames at 30 FPS), this extra latency can be absorbed by the buffer as long as

$$\delta_I < \delta_t - (max(\delta_{frame}) - min(\delta_{frame})) \quad (4)$$

with $max(\delta_{frame})$ and $min(\delta_{frame})$ being the maximum and minimum frame generation and transmission latency of the frames currently in the buffer $\delta_{frame} = \delta_{T,mess} + \delta_{T,frame} + \delta_R$.

In this model, we presume that the coverage area of edge servers is not limited to the base station [15], allowing us to consider the handover and migration as two separate events. As such, we disregard the handover's impact on migration.

4 IMPLEMENTATION

We implement Talaria as a plugin to the Unity engine that we integrate within the demonstration game provided with the 3D Game Kit plugin[5]. This game allows us to demonstrate the applicability of Talaria on an existing real-life polished game project while minimising the development effort on the game itself.

We first create an in-engine remote gaming system that forwards the input from the client to the server, receives the frames from the server and displays them on-screen. The client captures the user input and transmits them to the server. Upon reception of client input, the server updates the game state. The primary camera renders the corresponding frame into a RenderTexture, which is copied to the socket. The client copies the received frame onto a RenderTexture applied to a RawImage object to display it at the correct resolution.

Our implementation runs the game at 1920x1080 px and a framerate of 30 FPS. As this paper focuses on game instance migration rather than the implementation of a cloud gaming platform, we perform the following simplifications: (1) The server does not encode the frames, resulting in a significant network load; (2) Data is transmitted over TCP for reliability and in-order delivery at the transport layer; (3) The client operates on a best-effort basis and does not buffer the frames; (4) Many works already focus on mobility prediction. We thus trigger the preloaded instance setup and the migration through client key presses [20]. These simplifications present little impact in Talaria's operation, as discussed in Section 6.

Talaria relies on few elementary components. It is thus easily portable to other games, granted that the game state variables are declared public within their respective classes. Talaria requires synchronisable objects to be explicitly identified, which can be performed through a *Synchronisable* tag. However, the Unity Engine only allows a single tag per object, which may interfere with tags that have already been assigned in established projects. We thus rely on a *Synchronisable* Prefab to attach to the synchronisable game objects as a child. This prefab allows Talaria to programmatically find all synchronisable objects and assign them a unique ID without further developer intervention. This prefab is attached to all objects that can be altered during the game: the player character and camera, enemies, and destructibles. The demonstration game in the prototype implementation contains 51 synchronisable objects: the player character, the camera, 28 enemies and 23 inanimate destructibles.

A single script implements Talaria's operation as described in Section 3. Talaria saves and restores only the states that directly affect the gameplay to minimise the transmitted data. We thus focus on: (1) *Game Object's position and rotation*, (2) *Animator state*, (3) *Remaining health*. The enemies' behaviour state machine is represented by the Animator's state machine, their position, and the player character's position. As such, we do not need to synchronise behavioural data. The demonstration game in the prototype implementation relies on standardised elements, where the state of all objects is read and set through public variables and dedicated functions. Only the script that controls all damageables required to introduce a setter function for the remaining health, leading to a total of 10 lines of codes to modify in the game's codebase. In total, Talaria consists of a prefab and a single 400-lines script at the server-side and little adaptation to the existing project's codebase (10 lines in our example project). We represent the state of each Unity game object with a C# object that we serialise in the corresponding SYNC message. For the sake of simplicity, we do not consider movable objects that only fill a decorative function (vines, fireflies) and disable their motion.

We implement the synchronisation waves as follows: the first SYNC synchronises the player character and the camera; the second SYNC targets the visible objects; the third SYNC synchronises all remaining objects. The third SYNC does not contain the animator state, as the objects are not visible to the player. However, synchronising down to the exact animation frame is critical in previous waves to make the migration unnoticeable to the user and preserve the objects' state as Unity's animator drives the behaviour state machine.

5 EVALUATION

Evaluation setup: We run the two server instances and the client on a single computer running Windows 10 with CPU Ryzen 9 5900x, Nvidia RTX 3090, and 64GB RAM. Running all three components on the same machine allows us to focus on characterising the system migration independently from the network conditions. We start both server instances at the same time and trigger the initial instance setup on server B as well as the live migration through two distinct keystrokes at the client-side (see Section 4).

Pipeline Characterization: Figure 4 presents the end-to-end latency of the pipeline, from the transmission of user input to the display of the corresponding frame on the client. Using the Unity profiler, we select 10 random frames on the client and 10 random frames on server A and average the duration of each operation. On the client, capturing and transmitting user input takes 0.46 ms (std=0.065). The server then takes 5.00 ms (std=0.73) to execute the game logic, 4.46 ms (std=0.27) to render the frame, and 9.44 ms (std=3.26) to transmit the frame. The client receives the frame in

[5] https://assetstore.unity.com/packages/templates/tutorials/3d-game-kit-115747

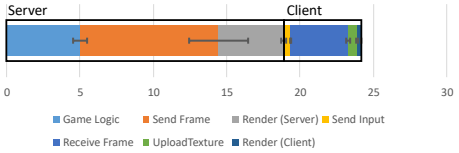
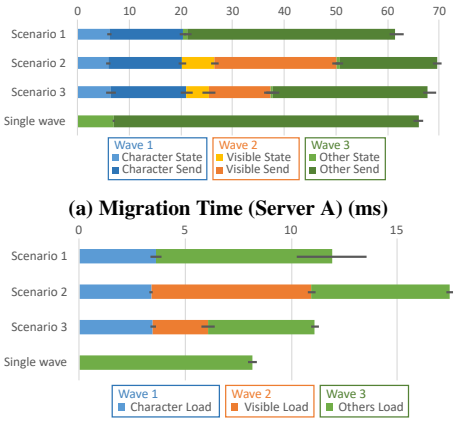


Figure 4: End-to-end latency of the prototype system's cloud gaming pipeline. The server process frames in 18.9 ms on average, while the client takes less than 5 ms to display.

(a) Scenario 1 **(b) Scenario 2** **(c) Scenario 3**

Figure 5: Experimental Scenarios



(a) Migration Time (Server A) (ms)

(b) Migration Time (Server B) (ms)

Figure 6: Migration time. With Talaria, the player starts the game on Server 2 at the end of the first wave (20 ms). Total migration takes less than 90 ms.

3.94 ms (std=0.21), uploads it to a texture in 0.45 ms (std=0.58), and renders in 0.30 ms (std=0.31). As we do not encode frames, sending and receiving the frame spreads over 13 ms. Encoding would bring down this delay at the cost of extra encoding latency. We thus consider this latency to be reasonable. The server processes frames in an average of 18.9 ms with an end-to-end latency below 25 ms, allowing us to run the game without buffering the frames.

We then characterise Talaria's migration. We measure the service downtime, the time for each wave, the total synchronisation time, and the network load and compare these measurements to a single-wave baseline. All measurements are averaged over 10 runs with 95% confidence interval error bars. We consider the following settings (see Figure 5): (1) start of the game, no synchronisable object in the player's view; (2) high point, many synchronisable elements visible in the distance (25 objects, 15 animated); (3) side of the map, between the first two scenarios (15 objects visible, 7 animated).

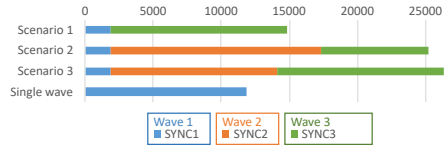


Figure 7: Network load (KB). Less than 30KB are transmitted over the network. Aggregating objects in waves significantly compresses the network load.

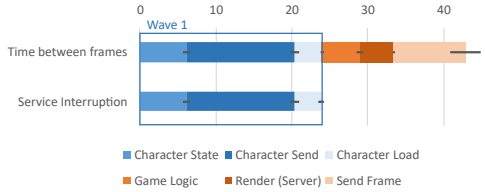


Figure 8: Time between the last frame from server A and the first frame of server B (top) and service interruption time (bottom) during the migration. The average interframe time is below 43 ms and the service interruption time below 25 ms.

Migration time: Figure 6 decomposes the migration time on server A (6.a) and server B (6.b). Loading the entire game state (all three SYNC messages) on server B takes less than 20 ms in all scenarios. On the other hand, animator synchronisation is costly. The total synchronisation time is proportional to the number of visible animated objects (about 0.27 ms per inanimate object and 1.90 ms per animated object). The time to identify objects as immediately relevant in wave 2 is negligible as we only consider objects visible on-screen. The total migration takes less than 90 ms in all scenarios.

Network load: The network load (Figure 7) is lightweight, with the entire migration fitting in less than 30 KB. Each migration wave consists of serialised state objects containing only the primary states variables necessary to preserve the current action. The camera object takes 796 B when serialised independently, while the player character synchronisation takes 1570 B. Wave 2 objects featuring an Animator component take 1441 B while objects without an Animator and wave 3 objects take 800 B. We aggregate these objects on a per-wave basis: camera + player character in wave 1, all visible objects in wave 2, and all invisible objects in wave 3. This aggregation reduces the network load by 25 to 50% depending on the number of aggregated objects by minimising the amount of data required for serialisation. Besides such aggregation, we do not perform any other compression.

Service Interruption: Figure 8 represents the time between the last frame from server A and the first frame from server B, and the service interruption duration. With Talaria requiring only the first SYNC packet for the client to switch server (wave 1), the service interruption time remains below 25 ms, leading to a total time between frames is below 44 ms. Given Equation 4, this latency can be made invisible as long as $\delta_t > 44 + (max(\delta_{frame}) - min(\delta_{frame}))$. For 100 ms buffering, this corresponds to 66 ms between frame arrivals, which is high even in current LTE networks. In comparison, single-wave migration takes over 74 ms, leading to 2 to 3 skipped frames, a noticeable interruption of the game display. Talaria results in a 61% shorter downtime than single-wave migration.

Table 1: Talaria vs state-of-the-art migration solutions

Solution	Chkpt. time	Restore time	Net. Load	Downtime
C/R [11] – Qwaq	5.6 s		N/A	4.2-5.6 s
C/R [11] – glxgears	8.6 s		N/A	6.45-8.6 s
C/R [8] – Maya	1.9–4 s		N/A	>1.9–4 s
C/R [8] – glxgears	2.16 s		N/A	>2.16 s
C/R [18] – openarena	2 s	17 s	165 KB	16.5–19 s
Live Migration [22]	4.82–23 s		>100 MB	6.8–25 s
Single-wave Talaria	**0.066 s**	**0.008 s**	**<30 KB**	0.074 s
Talaria	0.069 s	0.018 s	**<30 KB**	**0.025 s**

6 DISCUSSION

Comparison to traditional checkpointing: As stated in Section 2, few works target GPU checkpointing [8, 11, 18]. We compare Talaria to these works in Table 1. All solutions take several seconds to migrate light 3D applications. Besides, they rely on intercepting API calls, further impacting application performance [8, 18]. Traditional checkpoint-based CPU migration takes up to 7.8 s notwithstanding network transmission [22]. In comparison, Talaria leads to a service downtime below 25 ms, with a total migration time of 87 ms. Even when synchronising in a single wave, our method only takes 74 ms, two orders of magnitude faster than the existing methods. Lin et al. [11] raise the possibility of transmitting the OpenGL commands before the migration, decreasing the downtime by up to 25%. However, this approach still results in migration times of the order of the second while Talaria essentially nullifies the service interruption. Note that such performance is achieved with a more resource-intensive game than state-of-the-art approaches. At 1920x1080 px, openarena uses 108 MB RAM while the game used to demonstrate Talaria uses 764 MB. Even the Maya instance considered by Hou et al. is loaded with a light model (121 MB). Checkpointing our demonstration game would thus result in much larger files, leading to longer checkpointing duration and higher network latency.

Implementing Talaria in existing games: Talaria is designed to integrate existing Unity projects seamlessly and leverages standard Unity components to perform the migration. Developers can integrate Talaria in their projects by importing the package and adding the *Synchronisable* prefab to objects. If the project relies on Unity's standard scripts and Animator, the objects can be synchronised with no further intervention. We demonstrate this feature in Section 4 by integrating Talaria within an existing project seamlessly. If the project does not rely on Unity's standard components, it is necessary to declare the properties of the synchronisable objects explicitly in the code responsible for dumping and serialising the objects' states. Most games embed a save feature that performs similar operations, so the additional development cost would remain minimal.

Scaling to larger projects: In this paper, we implement Talaria over a fairly sophisticated example project. However, the most ambitious games developed with the Unity engine may present many more objects. In such a case, the developer may choose to add multiple migration waves to rebuild the state of the game over time. If many elements need to be synchronised during the first wave, a brief interruption may be acceptable if the migration is combined with game loading times (which would be possible by incorporating the migration with the engine). Given the current performance of Talaria, we estimate that the first wave would require to contain 1180 elements to present a service interruption over one second, still much lower than current state-of-the-art checkpointing solutions. The demonstration game is also a reasonably paced game, which is quite forgiving of user error. We expect faster-paced games such as first-person shooters to require further optimisation as even single-frame issues can impact the user experience.

Limitations: In-engine operation limits the applicability of Talaria to the goodwill of the developers. We address this issue by implementing Talaria as a Unity engine plugin that simplifies integration within existing projects. Besides, we expect edge gaming to be driven by major actors who have the resources to deploy an entire edge infrastructure and control over game development (potentially over their own game engines). In-engine mobile edge gaming would therefore represent a reasonable solution for such actors. Our implementation does not consider encoding/decoding and does not buffer frames to address potential jitter. Encoding frames would significantly reduce the network transmission time, at the cost of a short encoding and decoding latency (few ms per frame[6]). In-engine cloud gaming can also significantly accelerate encoding and decoding latencies by avoiding costly CPU-GPU copies. As stated in Sections 3.3 and 5, buffering frames would minimise service interruption and framerate drops during the migration.

Future Works: This study provides the first step towards ultrafast migration of edge gaming instances. We plan to address the limitations of our implementation and integrate frame encoding/decoding, lighter transport protocols, and frame buffering. We will also implement Talaria within a larger-scale first-person shooter game, featuring dynamic movement and many enemies. Such a game will allow us to control the game element synchronisation fully. We will consider what happens when many synchronisable objects are in the player's view and how to synchronise the game when the player can interact (shoot) with faraway objects. It will also allow measuring the impact of migration on the player's performance and satisfaction in fast-paced games. Finally, we will extend our experiments to confirm the operation of Talaria in a wide range of network conditions and characterise the impact of migration on user experience.

7 CONCLUSION

In this paper, we presented Talaria, a framework for ultrafast migration of edge gaming instances. By integrating the migration algorithm in the game's engine, Talaria results in near-seamless migration, with imperceptible service interruption and minimal impact on what the player sees on-screen. Talaria synchronises the essential gameplay elements between the two game instances faster and switches servers than the interframe time, preventing drops in framerates. As such, Talaria outperforms checkpoint-based state-of-the-art solutions by two orders of magnitude.

ACKNOWLEDGMENTS

This work was supported by the 5GEAR project (Decision No. 318927) and the FIT project (Decision No. 325570) funded by the Academy of Finland.

[6] https://developer.nvidia.com/nvidia-video-codec-sdk

REFERENCES

[1] Rami Akrem Addad, Diego Leonel Cadette Dutra, Miloud Bagaa, Tarik Taleb, and Hannu Flinck. 2018. Towards a Fast Service Migration in 5G. In *2018 IEEE Conference on Standards for Communications and Networking (CSCN)*. 1–6. https://doi.org/10.1109/CSCN.2018.8581836

[2] Hao Chen, Xu Zhang, Yiling Xu, Ju Ren, Jingtao Fan, Zhan Ma, and Wenjun Zhang. 2019. T-Gaming: A Cost-Efficient Cloud Gaming System at Scale. *IEEE Transactions on Parallel and Distributed Systems* 30, 12 (2019), 2849–2865. https://doi.org/10.1109/TPDS.2019.2922205

[3] K. Chen, Y. Chang, H. Hsu, D. Chen, C. Huang, and C. Hsu. 2014. On the Quality of Service of Cloud Gaming Systems. *IEEE Transactions on Multimedia* 16, 2 (2014), 480–495. https://doi.org/10.1109/TMM.2013.2291532

[4] Eduardo Cuervo, Alec Wolman, Landon P. Cox, Kiron Lebeck, Ali Razeen, Stefan Saroiu, and Madanlal Musuvathi. 2015. Kahawai: High-Quality Mobile Gaming Using GPU Offload. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services* (Florence, Italy) *(MobiSys '15)*. Association for Computing Machinery, New York, NY, USA, 121–135. https://doi.org/10.1145/2742647.2742657

[5] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. 2008. Parallel Computing Experiences with CUDA. *IEEE Micro* 28, 4 (2008), 13–27. https://doi.org/10.1109/MM.2008.57

[6] Keerthana Govindaraj and Alexander Artemenko. 2018. Container Live Migration for Latency Critical Industrial Applications on Edge Computing. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, Vol. 1. 83–90. https://doi.org/10.1109/ETFA.2018.8502659

[7] Kiryong Ha, Yoshihisa Abe, Zhuo Chen, Wenlu Hu, Brandon Amos, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2015. Adaptive VM handoff across cloudlets. *Technical Report CMU-CS-15-113* (2015).

[8] David Hou, Jun Gan, Yue Li, Younes El Idrissi Yazami, and Twinkle Jain. 2021. Transparent Checkpointing for OpenGL Applications on GPUs. arXiv:2103.04916 [cs.DC]

[9] Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Yury Degtyarev, Sergey Grizan, Alec Wolman, and Jason Flinn. 2015. Outatime: Using Speculation to Enable Low-Latency Continuous Interaction for Mobile Cloud Gaming. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services* (Florence, Italy) *(MobiSys '15)*. Association for Computing Machinery, New York, NY, USA, 151–165. https://doi.org/10.1145/2742647.2742656

[10] Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Yury Degtyarev, Sergey Grizan, Alec Wolman, and Jason Flinn. 2015. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. 151–165.

[11] Yunbiao Lin, Wei Wang, and Kyle Gui. 2010. OpenGL Application Live Migration with GPU Acceleration in Personal Cloud. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing* (Chicago, Illinois) *(HPDC '10)*. Association for Computing Machinery, New York, NY, USA, 280–283. https://doi.org/10.1145/1851476.1851510

[12] Fei Ma, Feng Liu, and Zhen Liu. 2010. Live virtual machine migration based on improved pre-copy approach. In *2010 IEEE International Conference on Software Engineering and Service Sciences*. 230–233. https://doi.org/10.1109/ICSESS.2010.5552416

[13] Lele Ma, Shanhe Yi, Nancy Carter, and Qun Li. 2019. Efficient Live Migration of Edge Services Leveraging Container Layered Storage. *IEEE Transactions on Mobile Computing* 18, 9 (2019), 2020–2033. https://doi.org/10.1109/TMC.2018.2871842

[14] Lele Ma, Shanhe Yi, and Qun Li. 2017. Efficient Service Handoff across Edge Servers via Docker Container Migration. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing* (San Jose, California) *(SEC '17)*. Association for Computing Machinery, New York, NY, USA, Article 11, 13 pages. https://doi.org/10.1145/3132211.3134460

[15] Francesco Malandrino, Scott Kirkpatrick, and Carla-Fabiana Chiasserini. 2016. How Close to the Edge? Delay/Utilization Trends in MEC. In *Proceedings of the 2016 ACM Workshop on Cloud-Assisted Networking* (Irvine, California, USA) *(CAN '16)*. Association for Computing Machinery, New York, NY, USA, 37–42. https://doi.org/10.1145/3010079.3010080

[16] Farouk Messaoudi, Adlen Ksentini, and Philippe Bertin. 2018. Toward a Mobile Gaming Based-Computation Offloading. In *2018 IEEE International Conference on Communications (ICC)*. 1–7. https://doi.org/10.1109/ICC.2018.8422518

[17] Aaftab Munshi. 2009. The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 1–314.

[18] Samaneh Kazemi Nafchi, Rohan Garg, and Gene Cooperman. 2014. Transparent Checkpoint-Restart for Hardware-Accelerated 3D Graphics. arXiv:1312.6650 [cs.OS]

[19] Javier Prades and Federico Silla. 2017. Turning GPUs into Floating Devices over the Cluster: The Beauty of GPU Migration. In *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*. 129–136. https://doi.org/10.1109/ICPPW.2017.30

[20] Zeineb Rejiba, Xavier Masip-Bruin, and Eva Marín-Tordera. 2019. A Survey on Mobility-Induced Service Migration in the Fog, Edge, and Related Computing Paradigms. *ACM Comput. Surv.* 52, 5, Article 90 (Sept. 2019), 33 pages. https://doi.org/10.1145/3326540

[21] Hiroyuki Takizawa, Katsuto Sato, Kazuhiko Komatsu, and Hiroaki Kobayashi. 2009. CheCUDA: A Checkpoint/Restart Tool for CUDA Applications. In *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*. 408–413. https://doi.org/10.1109/PDCAT.2009.78

[22] Roberto Torre, Elena Urbano, Hani Salah, Giang T. Nguyen, and Frank H. P. Fitzek. 2019. Towards a Better Understanding of Live Migration Performance with Docker Containers. In *European Wireless 2019; 25th European Wireless Conference*. 1–6.

[23] Shangguang Wang, Jinliang Xu, Ning Zhang, and Yujiong Liu. 2018. A Survey on Service Migration in Mobile Edge Computing. *IEEE Access* 6 (2018), 23511–23528. https://doi.org/10.1109/ACCESS.2018.2828102

[24] D. Wu, Z. Xue, and J. He. 2014. iCloudAccess: Cost-Effective Streaming of Video Games From the Cloud With Low Latency. *IEEE Transactions on Circuits and Systems for Video Technology* 24, 8 (2014), 1405–1416. https://doi.org/10.1109/TCSVT.2014.2302543

[25] Shucai Xiao, Pavan Balaji, James Dinan, Qian Zhu, Rajeev Thakur, Susan Coghlan, Heshan Lin, Gaojin Wen, Jue Hong, and Wu-chun Feng. 2012. Transparent Accelerator Migration in a Virtualized GPU Environment. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. 124–131. https://doi.org/10.1109/CCGrid.2012.26