DanceMark: An open telemetry framework for latency-sensitive real-time networked immersive experiences

Babis Koniaris* Edinburgh Napier University
David Sinclair† Edinburgh Napier University
Kenny Mitchell‡ Edinburgh Napier University

ABSTRACT

DanceMark is an open telemetry framework designed for latency-sensitive real-time networked immersive experiences, focusing on online dancing in virtual reality within the DanceGraph platform. The goal is to minimize end-to-end latency and enhance user experience, particularly crucial in partnered dancing scenarios where synchronized movements are integral. DanceMark supports experimentation addressing latency issues through direct sensor-to-display paths and real-time dance prediction correctives through offline analysis and online adaptive responses to live captured session data.

Index Terms: H.5.2 [User Interfaces]: User Interfaces—Graphical user interfaces (GUI); H.5.m [Information Interfaces and Presentation]: Miscellaneous C.2.m [Networks]: Network performance evaluation—

1 INTRODUCTION

We demonstrate the use of telemetry to measure the end-to-end latency as used in *DanceGraph* [18], a developing networking platform intended for online dancing in virtual reality. The project aims at providing an as-direct-as-possible networking architecture to mitigate latency to the maximum permissible extent, whilst simultaneously predicting users to sidestep latency issues that cannot be thus reduced. While latency is a factor in many real-time networked applications, with very definite consequences in user performance [15], it causes extra challenges with the issue of partnered dancing, where the time synchronization between user movement is an integral and ever-present feature of the application. Each dancer must locally perceive that their movements are in time with those of their remote partner, and this effective synchronization should persist throughout the entire experience.

The DanceGraph project mitigates the primary networked dance latency counter-measures, through the use of two features, a) reduced end-to-end latency by providing *as-direct-as-possible* paths from sensors to display and, b) Real-time dance correctives applied as rhythmic motion predictions to present multiple body motions synchronized with local music for each online dance partner's experience. For the first of these points, we must delineate the sources of latency and for the second provide an informed basis for predictions to align synchronized dance pose pairs. We call the framework of components providing this dance synchronization telemetry support, *DanceMark* (figure 1).

For local signals, such as dance pose streams from a tracking camera, we have a relatively straightforward task, since timestamps can be simply applied and dumped to a local file or output device. Latency measurements among networked clients, however, must coordinate among the network recipients all being on different machines at different locations, with relatively unimpeded signal packets transmitted over unidirectional protocols (UDP) with no acknowledgments or handshakes, and further suffering differing clock synchronizations on the various network nodes.

To this end, each DanceGraph client employs a configurable latency measurement system where servers collect bundled packets of latency measurements collated on designated clients via DanceGraph's network receivers.

Certain online dance networking signal types also facilitate extra latency tracking information to report client measurements, e.g., the latency introduced by body-tracking camera ingest processing times, and subsequent image processing performance information.

Our flexible open-source telemetry scheme permits both lightweight live monitoring of low-overhead statistics alongside deeper capture for aggregated multi-client offline analysis.

2 RELATED WORK

Social VR is an emerging field that uses virtual reality technology to create immersive digital environments where users, represented by customizable avatars, can interact with each other in real-time. It aims to simulate a shared physical space, fostering social interactions through activities, communication, and collaboration. Several social VR platforms exist, VRChat, Rec Room, and Facebook Horizon and some dance focused networked video games (including Dance Central and Just Dance), but also research lab originated platforms, like *Ubiq* [7] and *MOSIM* [8]. In particular, Ubiq is a toolkit that enables the creation of Social VR applications, and as such it focuses on openness and extensibility. However, scalability is still an issue [6], in addition to latency, especially when transferring/synchronizing a high volume of data, which might be necessary for streaming certain hardware signals such as immersive video streaming [4].

2.1 Latency

To a degree, latency in extended reality can be tolerated in terms of task performance but degrades the experience in terms of player experience quality and aesthetics [19]. As such latency is important in networked applications, and especially in interactive games. Jiang et al [10] show that players overall notice latencies as low as 50ms and can tolerate them up to 125ms, although there are variations across different genres, with FPS and subsequently XR games being more sensitive to latency than others (e.g. turn-based games).

Elbamby et al. [5] highlight latency as one of the potential adoption issues of networked VR applications over 5G networks and suggest solutions such as millimeter-wave communications, multi-access edge computing (MEC) caching and prediction, although they do not delve into specifics on the software side of these approaches. Chaccour et al. [3] suggest using the Terahertz band for wireless VR to provide a high-rate and high-reliability low-latency communication, which requires the software to make good use of such a networking architecture and sufficient availability of those hardware network services.

Jiang et al. [11] highlight the importance of identifying the sources of latency and identify and characterize different sources of delay in the networking infrastructure. Their work focuses on the networking infrastructure, and while VR (and music) is taken into account, as an application class with the lowest latency and highest data rate requirements, they do not expand on the software

*e-mail: b.koniaris@napier.ac.uk
†e-mail: d.sinclair@napier.ac.uk
‡e-mail: k.mitchell2@napier.ac.uk

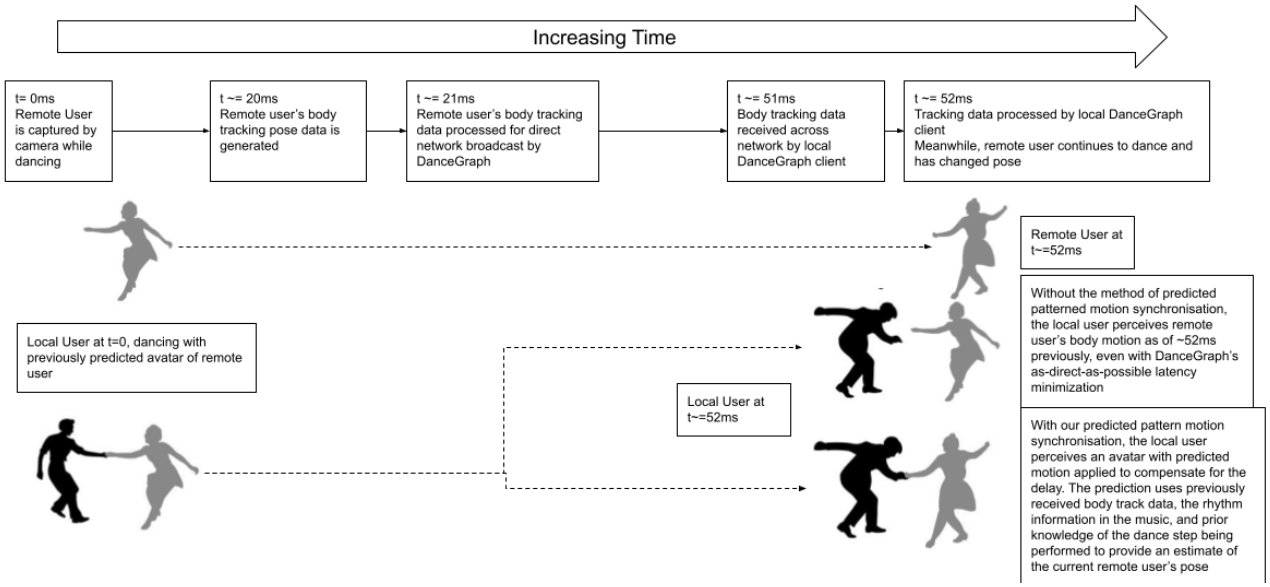


Figure 1: An illustration of the networking latency issues that DanceGraph attempts to solve

requirements or additional sensor data that can be used in immersive applications.

As networking infrastructure is not typically controlled by application developers, latency in software has to be mitigated with software-based methods. Liu et al. [14] developed a taxonomy for latency compensation techniques for games, and identified four main groups: feedback, prediction, time manipulation, and world adjustment, where most techniques (in terms of the number of publications) are based on prediction in the form of extrapolation. In the wider literature on motion prediction research often a large time window of sequences of poses or indeed the whole sequence is necessary for accurate results, whilst in the live networked case processing entire sequences is cost prohibitive and the stability of sequence frames can be erratic due to network conditions. In DanceGraph latency countermeasures for prediction are informed by DanceMark telemetry data capture operating within the *transformer* components of the architecture, e.g. a *rhythmic dance prediction transformer* [18].

2.2 Telemetry

Telemetry systems for interactive experiences are critical to not only improving the overall performance of the system throughout development, e.g. in multiplayer first-person shooters [17], but to informing with live data for adaptive real-time responses, as used in racing games [12].

The client-server telemetry architecture for the gathering latency timings was chosen since it mirrors and integrates with an already in-place architecture in DanceGraph [18]. Ajayi et al [1] demonstrate that using a publisher-subscriber model to collect data from Internet-of-Things applications provides for more efficient gathering in their use-case, but identified inefficiency as primarily due to the time cost of their client-server model keeping a dedicated connection open. Since DanceGraph's clients are constantly transmitting and receiving signal data to and from the server over the same network connection, DanceMark folds into this cost as part of normal operation.

DanceMark telemetry gathering is similar in spirit to the much more general approaches of Hyun et al [9] and Lin et al [13], where suitably equipped network nodes append timestamps to packets passing through them so that the end note has a complete record of the packet's latency profile.

3 Integrating DanceMark profiling in DanceGraph

DanceGraph is implemented as a server-client based system, with a thin server passing user signals (body tracking data, or microphone audio) and environment signals (global gameworld-related information, such as avatar appearances, usernames and so forth), between clients, as well as control signals, signals between server and client which primarily alter and update the state of the network. The use-case for DanceGraph is a non-competitive environment, so there is less emphasis on the server adjudicating a true common global state of the virtual world and its participants compared with a competitive networked game environment that ideally requires it. This further affords local dance pose malleability according to the ideal perceptual dance experience of each dancer local to their live body posture and flow in concert with the virtual representation of their remote partner.

Maximizing operational flexibility, the DanceMark attempts to decouple the generation, processing, and visualization of the various network signals it handles. Typical users will be connecting via a game engine, such as Unity or Unreal interfaced with DanceGraph, and viewing the results in a 3D gameworld on a Virtual Reality Headset.

DanceGraph's architecture is engine-agnostic with only a thin layer of glue code between the adaptor for the game engine and the network client software. Such adaptors may not necessarily reach out to the network, such as utility configurations that connect to a standalone motion capture client using shared memory inter-process communication (IPC). Integrated clients may indeed be applied to live link with digital content creation (DCCs) tools or headless machine learning frameworks with no inherent loss of performance. Small, command-line DanceGraph clients are also implemented for the purpose of testing and monitoring, as well as the recording and replaying of signal data for analysis and validation.

The decoupling of signal generation from signal propagation is implemented through the use of 'consumer' and 'producer' modules. 'Producers' are typically specific hardware drivers or network listeners that feed the signal data to the Signal Manager, the central

portion of the DanceGraph Client which routes the signal transport layer. 'Consumers' receive signals from the signal manager and dispatch them to the local system appropriately; a consumer can send a signal to a game engine for rendering, write signal data to a file, or dump information about the signal on the screen. Usually, though not always, a consumer will be agnostic to the exact type of signal being processed.

To illustrate this, a typical use-case for DanceGraph would be the transport of the pose information from a Stereolabs ZED 2[1] depth camera. Initially, the dancer is captured by the camera hardware, and a producer module called from the Signal Manager's main loop, would use the ZED SDK to obtain the image data and process it into a lightly compressed pose-tracked skeleton-data signal. The Signal Manager then passes pointers to the data to one or more consumers, including one to the adapter for the game engine, as well as to the network server, which transmits the pose data to a remote client. This remote client then dispatches the signal to the corresponding consumers for visualization and use by the remote user. In bypassing the game engine's dependencies on the ZED SDK and game engine's networking facilities, DanceGraph already allows for a significant reduction in latency due to local signal transport.

3.1 Multithreading

The DanceGraph architecture is extensible in terms of supported hardware signals, such as audio, haptics, camera, generated from different devices using bespoke bytestreams. In high-level terms, using 'producer' functions that utilize external libraries to generate signal data. Such signals can take a variable amount of time to generate, so it is not good practice for a signal producer to be dependent on other signal producers to complete. As such, signal producers operate in different threads, that are children of the native client thread. Typical hardware signals utilized in a client are expected to be well below the average number of hardware threads in a modern system, so there is no issue of thread starvation or suboptimal use of threads.

The native client also runs in its own thread, independent of the adapter endpoint (e.g Unity) that it is called from. This is essential for decoupling the update rates of networking and the endpoint application. The application is expected to run typically at 60Hz, and if the networking system is coupled to this rate, then we can only poll for data every 16ms. To prevent this artificial rate limiter, we decouple the execution of DanceGraph from the adapter application, via running DanceGraph in its own thread, with a user-defined update rate normally a the highest service rate of connected producer hardware sensors. As a result, the data is received at the DanceGraph client as soon as possible, whereupon other work can be done immediately (such as using transformers on received signals), without having to wait for the adapter endpoint's typically erratic and slower update function.

By using the telemetry system explained later in 3.2, we measure the latency of an artificially produced signal through DanceGraph to another local client at about 2.29 milliseconds, compared with the same types of signals being passed through DanceGraph while coupled to the engine tick rate at 6.88 milliseconds. Both engine tick and camera producer rates here are far from the worst-case; the body tracking function from the SDK of our ZED cameras can easily block the ZED pose producer for well over fifty milliseconds.

3.2 DanceMark Telemetry

The architecture supports telemetry of signal data, which is generally useful to have for debugging and general data analysis purposes. The data can be used to: measure the performance of the networking architecture, identify bottlenecks, track lost UDP messages, identify latency between clients, and for any other task that requires knowledge of a signal's journey from the source to all destinations.

[1] https://www.stereolabs.com/zed-2/

The telemetry system records each signal packet from its generation until consumed by any client. Such signal packet records include:

- **Client ID**: which client generated the signal.
- **Signal type**: what type of signal it is [body pose, voice, etc].
- **Packet ID**: a unique serial number for that signal type of the owning client.
- **Stage**: the stage in the signal journey.
- **Timestamp**: when did this packet ID arrive at this stage
- **Signal data**: optional data record, due to the additional requirements in storage and transfer.

The stages of the signal journey include *generation*, *sending to server*, *received at server*, *received at some native DanceGraph client* and *received at some client's adapter application*, etc. . For the client stages we make a distinction because the data will arrive as soon as possible at the native client, but the adapter application might cause additional latency due to having its own application update rate which might be different from the native client's update rate.

3.3 Telemetry Bundling

Sending telemetry data continuously can unnecessarily hurt network performance due to the additional load imposed by the extra data, therefore we need to be able to optionally enable and disable telemetry according to requirements. At the same time we need to be able to request data that have already been recorded. To solve this issue, we split the telemetry process into two parts: *service* and *request*. The *service* part runs continuously in all clients and server (if telemetry might be desired) and records locally, in memory, all required data. This is a very low-cost process as the records are small (if we don't request signal data) and the effect on performance is insignificant. The *request* part is executed on-demand: the server sends a message to all clients requesting telemetry data (this request could contain a filter for a time range, signal type, client subset or any other criteria). The clients, upon receipt of the message, start bundling their relevant recorded data into packets that are intermittently sent to the server. The server receives all data and integrates its own recorded data in order to produce a telemetry file (in our case, a simple CSV document) which can be analyzed at a later time (see Figure 3.5)

3.4 Timer Source of Truth

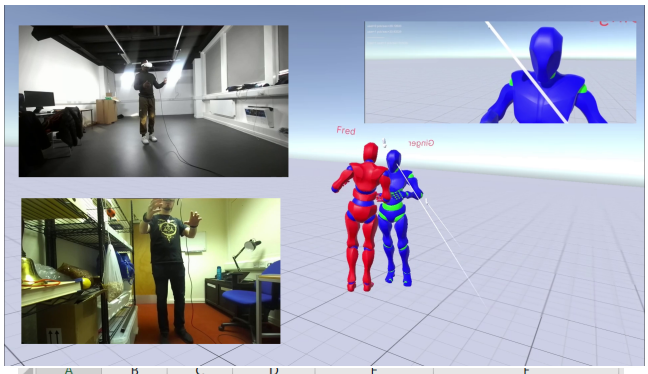
Telemetry data sourced from different clients contain timestamps using the local time for a client, which is typically not identical. In order to do meaningful data analysis we need as consistent date and time info among the server and clients as possible, and we achieve that (up to a small error) using NTP server queries. All clients and server measure their time offset to a per session, pre-set NTP server, calculated using an average offset from a number of queries executed when the server/clients start. When the clients connect to server, the server stores their time offsets, so that when signal timestamps arrive from clients as part of telemetry data, the server can offset the timestamps so that they are converted to the server's local time. This enables an *as-synchronous-as-possible* common frame of reference in terms of timing, that makes telemetry data analysis possible.

3.5 Live Adaptability

The capability of always-on low-cost recording of data with optional recording of signal data and on-demand requests for gathering data at the server allows the telemetry system to operate in different modes, a *runtime* mode where performance is prioritized, where signal data is not recorded but upon server request the data is collected for a specific set of parameters (specific clients, time range, signals), but also a *diagnostic* mode where everything can be recorded (including

signal data), enabling on-demand replay of the networked application as it's feasible to setup clients that can process incoming signal data based on telemetry diagnostic data.

Importantly, this also provides a framework for live adaption and animation of predicted remote motion poses to a reliable local time frame critical to experimentation with DanceGraph signal transformers (section 4.1).



	SigType	SigIdx	UserIdx	PacketId	TelemetryCapture	TimePoint
2	2	0	3	367	DataProduction	2023/11/08 23:13:42.91992
3	2	0	3	26	DataProduction	2023/11/08 23:13:35.06506
4	2	0	3	2	ReceivingAtServer	2023/11/08 23:13:34.53891
5	2	0	3	249	DataProduction	2023/11/08 23:13:40.18759
6	2	0	3	8	ReceivingAtServer	2023/11/08 23:13:34.67890
7	2	0	3	707	SendingToServer	2023/11/08 23:13:50.76592
8	2	0	3	1142	DataProduction	2023/11/08 23:14:00.74752
9	2	0	3	481	DataProduction	2023/11/08 23:13:45.54649
10	2	0	3	7	DataProduction	2023/11/08 23:13:34.62794
11	2	0	3	607	ReceivingAtServer	2023/11/08 23:13:48.48202
12	2	0	3	25	DataProduction	2023/11/08 23:13:35.04111
13	2	0	3	904	DataProduction	2023/11/08 23:13:55.29028
14	2	0	3	925	SendingToServer	2023/11/08 23:13:55.79571
15	2	0	3	1046	DataProduction	2023/11/08 23:13:58.54540
16	2	0	3	385	DataProduction	2023/11/08 23:13:43.33804
17	2	0	3	280	DataProduction	2023/11/08 23:13:40.90041
18	2	0	3	1	ReceivingAtServer	2023/11/08 23:13:34.51568
19	2	0	3	944	ReceivingAtServer	2023/11/08 23:13:56.22996
20	2	0	3	250	DataProduction	2023/11/08 23:13:40.20942
21	2	0	3	15	ReceivingAtServer	2023/11/08 23:13:34.84004
22	2	0	3	89	DataProduction	2023/11/08 23:13:36.52375
23	2	0	3	968	DataProduction	2023/11/08 23:13:56.74798
24	2	0	3	0	DataProduction	2023/11/08 23:13:34.46794
25	2	0	3	590	SendingToServer	2023/11/08 23:13:48.09809
26	2	0	3	822	SendingToServer	2023/11/08 23:13:53.42222
27	2	0	3	7	ReceivingAtServer	2023/11/08 23:13:34.65493
28	2	0	3	327	DataProduction	2023/11/08 23:13:41.99164
29	2	0	3	161	DataProduction	2023/11/08 23:13:38.16991
30	2	0	3	976	DataProduction	2023/11/08 23:13:56.92901

Figure 2: A pair dance instance of DanceGraph running a live Meringue server session between two clients and the raw telemetry data illustrated below.

4 Prototype Implementations

The flexibility of DanceMark integrated with the DanceGraph architecture allows the creation of prototype applications that can communicate with each other using the core networking component for setting up a server and the scene context for the clients: environment and related signals, and client-specific signals. Some of the different developed components include the native core library, different client signal libraries, server applications (GUI or headless), and client applications (native/C++ or Unity).

The native core library includes all the networking functionality required to set up a DanceGraph client/server environment. The different signal libraries are separate independent modules, compiled to DLLs that can be dynamically loaded by DanceGraph applications. We provide implementations for ZED camera, microphone data, and a number of test signals, with open adaptibility to MEMs sensor signals, haptics signals and beyond. The server application is used to distribute information, set up the environment and record telemetry, so it is developed as a C++ application, with headless and GUI variants. Client applications (adapter endpoints) are developed in a flexible way: as long as they link to a DanceGraph client DLL and use the exported API appropriately, the application can participate in the networked environment sessions. As such, we provide a Unity adapter that can use VR, ZED camera, and a number of other signals to visualize and participate in a scene (figure 3.5), but we also provide simple native adapters that can be used for testing, for example spawning a large number of "virtual" clients playing prerecorded animations in order to test performance and presence in group situations.

The 'producer' and 'consumer' abstractions for the DanceGraph Client provide natural boundaries where third parties can add modules to the project; these are implemented as dynamically linked libraries using C-style naming conventions with a small number of functions. Consumers and producers only require three functions each, with 'initialization' and 'shutdown' functions being called at the beginning and end of their lifecycle, respectively. Every tick, producers that are not busy are called and asked to place their signal data at a given memory pointer, and return the size of the data, and consumers will be provided with the pointer and data size. For users who wish to implement entirely new signal types, another 'config' DLL is required, which is called once at the initialization phase, and whose primary function is to inform the DanceGraph core software of the data sizes to facilitate appropriate memory allocations.

4.1 Interception and Modification of signals

The transformer section of DanceGraph, only briefly touched upon in prior literature, is the portion of the architecture which converts one or more signals, to another signal. This has a number of uses within the primary use-case for DanceGraph, where it can be used to predict remote user movements to mitigate latency, stylize body motion, or apply processing to incoming audio or outgoing video signals to reflect gameworld information.

The implementation of the transformer is that it is connected to the system as both a producer and a consumer module, with the associated API interfaces. Unfortunately, the naive API approach - that of having the transformer writer implement both the consumer-side and producer-side functions, has a couple of issues.

Firstly, individual producer calls take place in their own dedicated threads to avoid calls to hardware drivers blocking the main event loops in the game engine and network client, which in turn means that the 'consumer' API calls and the 'producer' API calls are inside different threads (see section 3.1) and signal information must be passed between them. This also means that every 'naive' transformer would contain significant amounts of very similar, and rather nontrivial code to do the job of passing consumed signal data to the producers.

To resolve this, DanceGraph's 'consumer' API calls are taken out of the hands of the module writer and are replaced by a class that stores a series of fixed-length queues of the relevant past signals, and which are accessible to the transformer via method calls.

4.2 Availability

The DanceGraph Software is open source, via two publicly available Github[2] repositories.

One repository contains the core DanceGraph software, written in C++20 for Windows 10 and above, which compiles using Visual Studio 22. The major external dependency for the core software will be the ZED SDK 4.0, with CUDA, which is an optional software driver required to facilitate body tracking using the ZED2 Depth Camera.

[2]https://github.com/CarouselDancing/DanceGraph

The other repository contains a Unity project with code and assets used to visualize the dancers; the repository's dependencies, other than larger project dependencies such as the Unity Client software itself, are contained within the project.

Further, various assets for environments are available open access [3]. As an open project community contributions are welcomed along with updates according to the CAROUSEL+ (EU grant No. 101017779) dancing online project developments and further future projects.

5 Conclusion

We show that our system designed for the transport of signals for online-dancing can seamlessly incorporate the use of telemetry for measurement of the recording of timings in network traffic for later statistical analysis and act as a framework for live adaption use cases.

6 Future work

DanceMark provides a framework for telemetry capture and use within the DanceGraph platform currently in development and there are a number of planned and potential continuations of the current work.

As DanceGraph operates in a fashion largely decoupled from the game engine used to display the application to the user, new game engine endpoints can be supported merely by implementing a thin adpator plugin to handle signal import and export, and support for engines such as Unreal or Godot and beyond.

With older works of analytic physics based prediction schemes [2] along with the rapid advances in machine learning for body tracking [16] give rise to the notion that state-of-the-art techniques for short-term motion prediction can be implemented as part of the prediction pipeline via transformers adapted to the challenge of real-time networked dancing partners.

The core component of the DanceGraph software is a generic low-latency signal transport agent, suggesting it's ability to be repurposed for other decentralized latency-sensitive, real-time multimodal networking applications, by implementing new modules for other use-cases.

Acknowledgments

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 101017779

References

[1] O. Ajayi, A. Bagula, J. Bode, and M. Damon. A comparison of publish-subscribe and client-server models for streaming iot telemetry data. In M. Masinde and A. Bagula, eds., *Emerging Technologies for Developing Countries*, pp. 129–139. Springer Nature Switzerland, Cham, 2023.

[2] S. Andrews, I. Huerta, T. Komura, L. Sigal, and K. Mitchell. Real-time physics-based motion capture with sparse sensors. In *Proceedings of the 13th European Conference on Visual Media Production (CVMP 2016)*, CVMP '16. Association for Computing Machinery, New York, NY, USA, 2016. doi: 10.1145/2998559.2998564

[3] C. Chaccour, M. N. Soorki, W. Saad, M. Bennis, and P. Popovski. Can terahertz provide high-rate reliable low-latency communications for wireless vr? *IEEE Internet of Things Journal*, 9(12):9712–9729, 2022. doi: 10.1109/JIOT.2022.3142674

[4] F.-Y. Chao, C. Ozcinar, and A. Smolic. Privacy-preserving viewport prediction using federated learning for 360° live video streaming. In *2022 IEEE 24th International Workshop on Multimedia Signal Processing (MMSP)*, pp. 1–6, 2022. doi: 10.1109/MMSP55362.2022.9950044

[5] M. S. Elbamby, C. Perfecto, M. Bennis, and K. Doppler. Toward low-latency and ultra-reliable virtual reality. *IEEE Network*, 32(2):78–84, 2018. doi: 10.1109/MNET.2018.1700268

[6] S. Friston, O. Olkkonen, B. Congdon, and A. Steed. Exploring server-centric scalability for social vr. In *2023 IEEE/ACM 27th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pp. 56–65. IEEE, 2023.

[7] S. J. Friston, B. J. Congdon, D. Swapp, L. Izzouzi, K. Brandstätter, D. Archer, O. Olkkonen, F. J. Thiel, and A. Steed. Ubiq: A system to build flexible social virtual reality experiences. In *Proceedings of the 27th ACM Symposium on Virtual Reality Software and Technology*, pp. 1–11, 2021.

[8] F. Gaisbauer. MOSIM end-to-end digital integration based on modular simulation of natural human motions. Technical report, 09 2019.

[9] J. Hyun, N. Van Tu, J.-H. Yoo, and J. W.-K. Hong. Real-time and fine-grained network monitoring using in-band network telemetry. *International Journal of Network Management*, 29(6):e2080, 2019. e2080 nem.2080. doi: 10.1002/nem.2080

[10] C. Jiang, A. Kundu, S. Liu, R. Salay, X. Xu, and M. Claypool. A survey of player opinions of network latency in online games. 2020.

[11] X. Jiang, H. Shokri-Ghadikolaei, G. Fodor, E. Modiano, Z. Pang, M. Zorzi, and C. Fischione. Low-latency networking: Where latency lurks and how to tame it. *Proceedings of the IEEE*, 107(2):280–306, 2019. doi: 10.1109/JPROC.2018.2863960

[12] E. Jimenez, K. Mitchell, and F. Seron. Capture and analysis of racing gameplay metrics. *IEEE software*, 28(5):46–52, 2011.

[13] W.-H. Lin, W.-X. Liu, G.-F. Chen, S. Wu, J.-J. Fu, X. Liang, S. Ling, and Z.-T. Chen. Network telemetry by observing and recording on programmable data plane. In *2021 IFIP Networking Conference (IFIP Networking)*, pp. 1–6. IEEE, 2021.

[14] S. Liu, X. Xu, and M. Claypool. A survey and taxonomy of latency compensation techniques for network computer games. *ACM Comput. Surv.*, 54(11s), sep 2022. doi: 10.1145/3519023

[15] I. S. MacKenzie and C. Ware. Lag as a determinant of human performance in interactive systems. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, CHI '93, p. 488–493. Association for Computing Machinery, New York, NY, USA, 1993. doi: 10.1145/169059.169431

[16] L. Mourot, L. Hoyet, F. Le Clerc, F. Schnitzler, and P. Hellier. A survey on deep learning for skeleton-based human animation. In *Computer Graphics Forum*, vol. 41, pp. 122–157. Wiley Online Library, 2022.

[17] J. Munro, K. Appiah, and P. Dickinson. Investigating informative performance metrics for a multicore game world server. *Entertainment Computing*, 5(1):1–17, 2014.

[18] D. Sinclair, A. V. Ademola, B. Koniaris, and K. Mitchell. DanceGraph: A complementary architecture for synchronous dancing online. *Short Paper presented at 36th International Computer Animation Social Agents (CASA) 2023, Limassol, Cyprus*, 2023.

[19] F. Zünd, M. Lancelle, M. Ryffel, R. W. Sumner, K. Mitchell, and M. Gross. Influence of animated reality mixing techniques on user experience. In *Proceedings of the 7th International Conference on Motion in Games*, MIG '14, p. 125–132. Association for Computing Machinery, New York, NY, USA, 2018. doi: 10.1145/2668084.2668088

[3] https://github.com/CarouselDancing/CarouselTangoWorld