Metaverse as a Service: Megascale Social 3D on the Cloud

Andreas Haeberlen
Roblox / University of Pennsylvania

Linh Thi Xuan Phan
Roblox / University of Pennsylvania

Morgan McGuire
Roblox

ABSTRACT

We present a vision for the future of an emerging category of cloud service: the metaverse of 3D virtual worlds. Today, hundreds of millions of users are active daily in such worlds, but they are partitioned into small groups of at most a few hundred players. Each group joins a different virtual world instance, and players can only interact in 3D with others players in the same group during that session. Current platforms are designed in ways that simply *cannot scale* much further, and solutions from other cloud services do not generalize to the more interactive, bidirectional, and latency-sensitive interactive 3D domain. We outline some of the technical challenges that currently stand in the way of a metaverse without inherent technical limitations on the number of users in a shared experience. We argue that, although these obviously touch on many other areas of Computer Science such as computer graphics and numerical simulation, the core challenges lie squarely within the systems domain.

CCS CONCEPTS

• **Networks** → **Cloud computing**.

1 INTRODUCTION

Currently, there is a lot of excitement about the vision of the *metaverse* as originally described by Stephenson [67] – a giant, immersive 3D virtual world with users interacting with each other through avatars, multiple linked virtual user-generated environments, and a digital economy making the entire system viable. There are many organizations building technology that aspires to this vision, and a major social network operator has even gone as far as renaming the entire company to reflect its commitment to it.

Making the metaverse a reality will require major advances in 3D computer graphics, virtual and augmented reality, physical simulation, and artificial intelligence. The corresponding research efforts are well underway and are receiving a lot of public attention. But, perhaps less obviously, the metaverse is also a prime candidate for a next-generation cloud service. We claim that the primary challenges to building the metaverse are in fact systems challenges in satisfying technical requirements common to 3D social interaction: low-latency communication, computation on heterogeneous client devices, synchronized distributed state, and partitioning and balancing the workload of that distributed system.

For example, we observe that current platforms in this space, such as Roblox, Fortnite (UEFN), Minecraft, and Second Life all demonstrate 1) a variety of different, linked experiences, which in current practice range from theme parks to concerts to massive battles, and 2) player creation of this content, independent of the vendor who provided the platform. Because of this, a separation emerges between metaverse-like *platforms* that provide the infrastructure for virtual worlds, and *content creators* that build and operate specific experiences on these platforms. Those content creators vary from individual amateurs to 200-person companies. This is roughly analogous to the platform-as-a-service model that is common on cloud platforms today, so rather than being a primarily game-like or VR-like technology stack, we see the metaverse as foremost a *real-time, multiuser* analog of Amazon Web Services (AWS) where each experience executes within a container-like isolation structure on abstracted data center resources. That they are 3D and maybe VR on the client is secondary to the real-time, multiuser, and multiple-application platform service aspects.

Because of these systems challenges, today's platforms fall short of the full metaverse vision because *they do not scale*! This may sound surprising, given that these platforms have tens of millions of active users each day. But in fact, the current platforms separate those players into millions of separate world instances (called "servers" in their user interfaces) that each contain a small number of users. Each user can only interact with the users on the same instance. The exact limit of players per instance varies between platforms: Minecraft becomes unusable after 400 users [73], Roblox is currently API-limited to 700 users [12], Second Life and Fortnite are limited to 100, and Horizon Worlds to 20 [6]. Some platforms support inter-instance mobility – perhaps by allowing players to "teleport" between instances – but it is always explicit and player-visible. Except for some carefully engineered one-off experiences that did not have general

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0387-4/23/11…$15.00
https://doi.org/10.1145/3620678.3624662

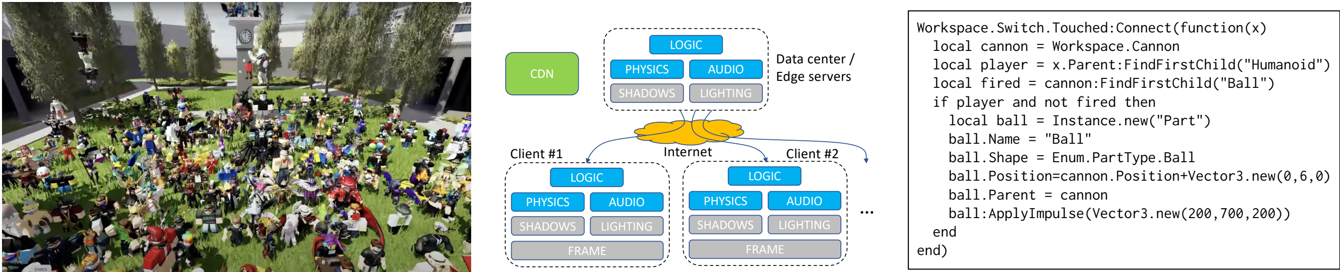

Figure 1: An scene from a Roblox experience (a), a sketch of the current architecture (b), and a simple script (c).

purpose interaction [70], we are not aware of *any* current platform that can support a real-time social 3D experience with more than 1,000 mutually visible and interacting users; a tiny fraction of the tens of millions online on these platforms today, and the billions of potential users.

The reason for this lack of real scalability is a combination of hard technical problems that have yet to be solved convincingly. In this paper, we sketch these problems, along with some potential solutions, and we outline a vision of a cloud-like metaverse platform that could truly achieve Stephenson's vision of a single shared world with tens of thousands (or perhaps even millions!) of users. We use the Roblox platform as a case study, and we argue that its underlying computation is quite different from pretty much every other current cloud service we are aware of: it comes with strict latency requirements, an unusual type of consistency, an extreme level of dynamism, new kinds of security risks, and many other interesting challenges.

This paper is a call to arms: although virtual worlds have not yet received a lot of attention from the systems community, we believe that there are many exciting problems to be solved, and that our community has a critical role to play!

2 OVERVIEW

Roblox is a popular online platform with about 66 million daily active users [61]. It is not a specific game by itself; rather, it is a platform that hosts a large number of 3D virtual worlds called *experiences*. Those can be created and published by anyone, using a freely available editor, Roblox Studio. We show a screenshot from an example experience in Figure 1(a).

Roblox users can assume one of two distinct roles: they can act as *creators* and make their own experiences, or they can act as *players* and join experiences that others have created. The platform itself is free to use; Roblox supports it by selling an virtual currency called Robux. Creators can also use their experiences to earn money in various ways, e.g., by offering virtual items for sale [60].

In Figure 1(b), we illustrate the architecture of the current Roblox platform [44, 53]. Roblox consists of two main functional components: a *simulation* that maintains and evolves the state of the virtual world, and a *renderer* that displays a (typically different) view of the state to each player. The simulation consists of the *game logic*, which describes how the world works, and a *physics* module that evolves object state based on the laws of physics. The game logic is expressed as *scripts* in a language called Luau [59]. The renderer contains a 3D audio component that produces an audio stream for each player, e.g., by mixing sound effects and voices of other nearby players, as well as the video computation, which roughly consists of shadows, lighting, and frame assembly.

The Roblox engine maintains two kinds of data: the *data model*, which is a dynamic, DOM-like tree that consists of object states and object positions, and *assets*, which are static descriptions of the objects themselves, such as 3D meshes, sounds, and Luau scripts. Since the assets are currently immutable (or at most change once in a few days), clients simply download them from a CDN on demand.

Figure 1(c) shows an example script that causes a cannon to emit a cannonball when the player touches a switch. The cannon and the switch are accessible via named vertexes in the data model, and they have properties (such as `Position`) and events (such as `Touched`). New data-model vertexes can be created at runtime, like our example script does to make the cannonball appear, and these changes have an immediate effect on the simulation. Once created, the cannonball is moved on a ballistic trajectory by the physics module.

A traditional game engine would run the simulation on the server and simply stream updates to client-side renderers. These renderers might do a bit of simulation on their own, e.g., by extrapolating object positions via dead reckoning to hide network latencies, but only the server's state is *authoritative*: if the client's extrapolation is wrong, it is overridden by the server's state. This approach can scale to tens of players, which is why most online platforms games are capped at 100 or fewer players per instance. In contrast, Roblox distributes game logic and physics across the server and the clients: a player's client can take over the authoritative simulation of objects that are close to that player. This reduces the load on the server and enables the client to tolerate even seconds of network latencies without user-perceived lag. With this approach, Roblox can scale each *instance* of an experience to about 700 players [12].

3 TECHNICAL CHALLENGES

Why can't a Roblox-like platform scale to experiences with more users? In this section, we discuss some of the technical challenges that currently prevent this.

3.1 Workload scalability

At first glance, simulating virtual worlds is an $O(N^2)$ task, since each of the N users can potentially interact with every other user, both directly through voice and virtual physics, and indirectly through effects caused by their actions. This would make virtual worlds inherently unscalable.

One way to prevent this computational explosion is to ensure that each object can interact with at most a limited number of other objects – ideally $O(1)$. Fortunately, virtual-world workloads have a property that naturally creates such a limit in most cases: they are *spatial*, in the sense that each object has a specific location in 2D/3D space, and they exhibit *spatial locality*: objects tend to interact mostly with objects that are close to them. This is certainly true of the physics simulation: current physics engines focus on mechanical properties, such as torque, momentum, velocity, etc., that change only when objects come in direct contact. Thus, the physics workload, which is a major part of the overall computation, is already $O(N)$.

Spatial locality is less obvious for scripts. With the current Roblox API, any script can access any object (by traversing the data model), and there are several non-local operations: for instance, a `DescendantAdded` event handler near the root of the data model would fire whenever an object is created *anywhere* in the virtual world, and scripts can invoke raycasts to find the closest object in a given direction, no matter how far away it is. Fortunately, there *is* a lot of locality in what scripts actually do: many scripts are attached to a specific object and interact only with that object (say, making a car turn when a player turns the steering wheel), or they are equivalent to such scripts: a script that implements the steering wheels of *all* the cars in the simulation is morally equivalent to just attaching the same script to each car.

Locality could be increased by removing unscalable primitives (such as the event handler in our example) and/or replacing them with scalable alternatives (such as a range-limited raycast). Perhaps each script could even be restricted to a specific object, or be given a "bounding box" that limits its data-model access to objects within this box. Functionality that is truly global, such as a Harry-Potter-style "Marauder's Map" that displays the current locations of other players anywhere in the virtual world, could be implemented with multiple scripts and explicit message passing.

Key challenges: The biggest challenge here is to achieve scalability without losing usability in the process. Defining a scripting language with perfect locality is not hard; defining one that is also intuitive, easy to learn, and can support a variety of complex experiences is much harder. (This is particularly true for a platform like Roblox, where many creators write their very first programs.) However, the community is no stranger to this kind of challenge; recall, e.g., the debate about the right language for data analytics [17, 18], distributed systems [36, 43], or privacy [7, 22, 46].

3.2 Concurrency

A scalable workload does not guarantee scalability by itself: since the work for a given frame must be completed before the start of the next frame (within 16ms, at 60fps), the only hope is to do as much of the work as possible in parallel. This is reasonably easy for physics, but not for scripts.

Achieving high concurrency basically requires two things: (1) as few dependencies between tasks as possible, and (2) a way for the platform to *recognize* the absence of such dependencies. In principle, metaverse workloads should have few dependencies: if one script operates a door and another the engine of a car, the two should rarely, if ever, need to interact. However, dependencies are sometimes created artificially by the API – e.g., by a promise that scripts will execute one at a time, or that events will be delivered in a certain order. This is convenient and easy to implement a single server, but it comes at a high price in a distributed environment. Dependencies can also arise from the way certain operations are defined: for instance, if objects are moved by setting velocities, any attempt by two scripts to move the same object will create a write-write conflict, whereas, if objects are moved by applying impulses, the impulses can simply be added up.

Even when there morally are no dependencies, the platform cannot necessarily be sure of this. For instance, two scripts that traverse the data model *could* potentially interact, and it is not always easy to prove the contrary, e.g., via static analysis. However, explicitly parallel primitives could help with this – perhaps a `forEach` loop that applies a certain operation to each element of a set, or each data-model vertex of a particular type. Such operations could easily be vectorized and/or run in parallel on different servers.

Key challenges: Increasing concurrency raises the spectre of race conditions and deadlocks, but careful API design [4, 15] could help with this – and perhaps also in the special nature of this workload. Maybe in this domain it makes more sense to lock space instead of state?

3.3 Distributed execution

With tens of thousands of players, even an $O(N)$ workload would quickly exceed the capacity of a single server and would thus have to be divided up among multiple machines. This raises two questions: (1) should these machines be infrastructure servers or player devices?, and (2) how exactly should the workload be divided up?

The answer to the first question is most likely "both". At the very least, the player devices would have to simulate the objects closest to their player's avatar – without this, it could take an entire WAN RTT, e.g., from the time a player kicks a ball until the ball responds. But true scalability would require more than that: the player devices would have to help with the heavy workload, otherwise the system would remain bottlenecked by the infrastructure servers. This is already standard in many multiplayer games [80], and it provides organic scalability [9, 63]: adding more players increases the workload but also adds resources with which to do the work.

However, some parts of the simulation would have to remain on the infrastructure, for several reasons. One is cheating [33]: if user devices are authoritative for part of the simulation and nobody checks their results, a user with a hacked device could open a locked door simply by setting the lock's state to 'open'. Others include safety (what if a hacked device is used to arrange objects in an obscene pattern?), user experience (overloading a mobile device with computations could drain its battery), and reliability: some players could have limited connectivity or suddenly switch off their devices, and this should not affect other players.

Overall, for both technical and economic reasons, we expect that the simulation would have to be distributed across a large number of infrastructure servers *and* player devices. This is a major difference to most existing cloud services, which run exclusively on infrastructure servers.

Key challenges: Coordinating a massive number of globally distributed devices is an obvious challenge, as are security and reliability. The extensive literature on peer-to-peer systems [37, 63] and especially peer-assisted systems [54, 83] could help. However, the existing solutions typically cannot support the stringent timing requirements of virtual worlds, so this is by no means a solved problem.

3.4 Partitioning

The next question is *how* the simulation should be divided up among these nodes. As with any other distributed system, the network will limit how much, and how quickly, the nodes can communicate, so it makes sense to keep most of the computation local. For instance, if a car consists of several connected parts (wheels, suspension, seats, etc.), the physics module needs to solve a complex system of equations to compute the state of these parts for each new frame [45, §15]. If the parts are stored on different machines, this requires an enormous amount of communication.

At first glance, one could simply divide up the simulation into groups of dependent objects – "physics islands" of connected parts, objects that are accessed by the same script, etc. – and assign each group to a specific node. But players and objects can move over time, so it is not that simple: a player might jump onto a moving train car and thus become part of the train car's physics island, and some computations, such as collision detection, have to reach across islands.

Additionally, the scale and distribution of objects and their interactions will vary across different experiences by different creators. Any static, uniform partitioning that uses virtual 3D position (such as a grid) as a heuristic of data dependency will have overloaded and underloaded elements, and any dynamic partitioning using off-the-shelf 3D data structures, such as BSP trees or Voronoi partitions, requires continuous updates that must themselves be synchronized. Simulation also is not necessarily solely collision response between objects. It encompasses fluid dynamics such as water and air, electronic and hydraulic simulation, realistic (e.g., gravity, magnetism) and fantastic (tractor beam, laser rifle, telekinesis) forces that operate over distances, and arbitrary programmable callbacks injected by the experience creator.

In some sense, the computation that advances the world state (through realistic physics or otherwise) is a data-flow graph that connects nodes representing elements of that state. A good partitioning of this graph is a form of min-cut that minimizes communication between the separated graph components, so that they can be assigned to different nodes. The components themselves can have dense computational dependencies. This is a generalization of the traditional concept of physics islands: instead of virtual 3D locality as a heuristic, the actual data flow dictates the partitioning.

Key challenges: Spreading a large workload with lots of dependencies across nodes is known to be a hard problem, as, e.g., the literature on graph processing [24, 82] can attest, but the spatial structure of metaverse workloads should help.

3.5 Load balancing

If the virtual world were static, the partitions could simply be assigned to nodes at the beginning and remain constant after that. However, in practice, players and objects will move around and can congregate in specific regions, e.g., for a large battle; also, players can join or leave, and objects can be created or destroyed. Thus, the load can grow or shrink, and it can become imbalanced across nodes. This suggests mechanism for shifting load from one node to another.

One approach would be to provide a mechanism for splitting and merging regions at runtime. For instance, if a concert venue fills up with guests, a piece of its partition could be split off and assigned to an additional node. However, as players and objects move around, frequent splitting could lead to complicated bookkeeping and a substantial amount fragmentation. It seems more promising to divide the world into fixed-size regions of virtual space – say, cube "tiles" of perhaps 100m to a side – and to assign sets of tiles to specific nodes [13]. This is roughly analogous to how operating systems manage virtual memory at the granularity of fixed-size pages and use a page table to control the location of each.

Notice that we have been carefully referring to "nodes" and not to servers in a data center: Roblox today distributes computation primarily to *client* nodes, albeit with a 3D spatial heuristic instead of a more sophisticated partitioning system like this. Also, the subdivision would not have to be exact: for the reasons discussed in Section 3.4, a train car that moves from a tile on server A to an adjacent tile on server B should be moved as a whole, and could thus be temporarily "sticking out of" the first tile. This is fine: the tiles would be used mostly for load management and their boundaries would not constrain directly physics and scripting.

Key challenges: Due to the stringent timing requirements of virtual worlds, "downtime" must be avoided during tile migrations. This problem has been solved, e.g., for live migration of virtual machines [14], and similar solutions could work here. The answer to the question *which* tile to move *when* is less clear; morally, this is a scheduling problem, but it is unlike any other we have seen in the literature.

3.6 Synchrony

In a metaverse, the players would naturally expect the simulation to be synchronous: if one player does something, other nearby players should see the effects immediately. However, in practice this is difficult to achieve because two players could be close in the virtual world but very far apart in the real world. Thus, the real-world laws of physics will limit how quickly information from one player's device can reach the other player's device.

For instance, consider the following scenario: Alice is in New York, Bob in Tokyo, and they are both connected to a game server in London. The one-way propagation delays are roughly 40ms from Alice to the server, and roughly 110ms from the server to Bob. In the virtual world, Alice is standing on a platform while a train with Bob on board is passing through at 22mph ($10\frac{m}{s}$). Alice is holding a ball, Bob briefly opens his window, and Alice throws the ball to Bob.

In a naïve implementation, the simulation runs on the server, and the server regularly sends updates to Alice's and Bob's machines. In this case, Alice will see the train with a 40ms delay, so, if she aims at the window where she sees it, the window will already have moved 40cm. Moreover, Alice's command to throw the ball will not reach the server until 40ms later, at which point the window will have moved another 40cm. Thus, Alice will most likely miss the window. With client-side execution (Section 3.3), the situation is even worse: the train car may be simulated on Bob's machine at the time, which would increase the RTT to 300ms.

A more attractive solution would be to run the simulation on a shared virtual timeline, so that Alice, Bob, and the server all see the train in the exact same location, and Alice's throw will hit the window. However, this works only if all the one-way delays are smaller than the inter-frame delay (33ms at 30fps). If some one-way delays are larger, as in our scenario, there is simply no way that an event on one machine could reach all the other machines in time for the next frame.

A common solution to the above problem is a predictive contract mechanism, such as "dead reckoning" [52]: Alice's machine can extrapolate the train's likely current position, and correct it once the next update arrives. However, simple extrapolation works only for position data, and even there it can produce serious artifacts: if the train travels on a narrow curve, linear dead reckoning will cause an ugly sawtooth pattern of short linear segments followed by corrections.

Key challenges: The key problem – presenting a consistent view to different players – has been studied extensively in the context of multiplayer games [42], but the solutions often rely on domain knowledge: for instance, to avoid player frustration, a first-person shooter game might "favor the shooter" and count a hit based on what the shooter saw, even if the server's view was different. A metaverse that is built on user-generated content would need a more general solution that works across many different kinds of experiences.

3.7 Speculation

The biggest problem with dead reckoning is its fairly simplistic predictions: unless objects happen to actually travel on a linear trajectory, frequent corrections are inevitable. Moreover, the only way to prevent large (and noticeable) corrections is for the server to send updates frequently, which consumes a lot of bandwidth.

One way to avoid this would be to leverage the entire game logic, instead of "just" the physics: nodes could be given replicas of entire tiles (scripts and all) and use *speculation* – a time-tested latency-hiding technique [8, 16, 49] – to roll forward the state of their replicas to the current virtual time, using the normal rules (physics, scripts, etc.) that apply to the entire simulation. In our above example, this would cause the train and the ball to move on their normal trajectories. Importantly, it would also work in more complex cases – say, if a script causes the ball to pop once it hits a sharp object.

Of course, nondeterministic events, such as player actions, can cause speculation to fail. For instance, while Alice is holding the ball in her hand, Bob's machine will predict that the ball remains where it is. Thus, when Bob's machine learns that Alice has thrown the ball, it needs to make a correction – but after that, the ball once again has the correct state and can be simulated locally, without further updates from Alice. Absent such nondeterministic events, replica tiles will evolve almost exactly as the original [20, 39], although occasional updates are still useful, e.g., because of FPU differences.

Key challenges: One obvious challenge is to guess correctly much of the time. Of course, this also applies to other forms of speculation, such as prefetching [3, 38] or speculative execution [35, 49], but good speculation often involves domain

knowledge, which may be hard to gather in a metaverse that consists of many different, user-generated experiences. Another, deeper challenge is to speculate efficiently across subsystems. Although some speculation is already done by modern game engines [21, 56], it is typically limited to physics state and does not involve complex, user-generated scripts.

3.8 Reconciliation

In general, it is not sufficient to correct just the single object that was affected by the nondeterministic event. For instance, suppose a pebble flies towards Alice when she is about to throw the ball. Bob's machine may speculate that no additional events will happen and keep the ball where it is, so the pebble may bounce off it. Once Bob's machine learns of the throw, it will have to "roll back time" analogous to [48], put the pebble back where it was, and redo the physics computation. This time, the pebble will continue on its trajectory, since there is no longer a ball it can bounce off of.

Speculation generally works well if 1) its guesses are mostly correct, and 2) being wrong is reasonably cheap. We hypothesize that 1) is true because speculation errors are driven by nondeterministic events, specifically user actions, which happen at human timescales: most users cannot push keys more than maybe twice per second, and often don't push any at all. 2) should be true because many events will affect only one physics islands' worth of computation (say, the position of the ball); few events would truly affect an entire tile. One could use provenance [10] to track dependencies and to decide what, if anything, needs to be recomputed, roughly analogous to view maintenance in databases [25].

One concern are misprediction cascades: what if the engine of Bob's train slows down and causes mispredictions along the entire train? One way to avoid this could be to use hierarchical simulation, that is, to first simulate a heavily simplified version of the train as a whole, and to then propagate the solution to a more detailed simulation of the individual cars. Another, related concern is that updates after failed speculation could cause visual artifacts. However, at least for physics state, this problem exists even with simpler approaches, such as dead reckoning, and there are existing solutions, such as interpolation techniques that slowly move objects from where they are to where they are supposed to be, potentially across several frames. The problem is harder for discrete state, though, such as events or variables in scripts.
Key challenges: In addition to the challenge of being right as often as possible (Section 3.7), it is important to minimize the cost of being wrong. Incremental updates could be the key to this: one could think of the simulation state as basically a materialized view over local and remote input events, with the latter changing over time as the local client becomes aware of past remote events on other clients. Thus, efficient view maintenance techniques [26] could be useful.

3.9 Consistency

The above discussion raises a larger question: how close do the virtual worlds on the different machines need to be? Keeping them exactly identical is impossible, because of propagation delays, and even very close consistency could be expensive. However, exact consistency is also unnecessary: the simulation is shown to human players, so small inconsistencies are okay if most players will not notice them.

And yet, *some* limit on the inconsistencies is clearly necessary: the train needs to be approximately in the right place, otherwise Alice might find herself run over if she attempts to cross the tracks. On the one hand, eventual consistency is not enough, since this would permit the platform to adjust the position of the train, say, tomorrow – long after it has already hit Alice. On the other hand, sequential consistency is unrealistic, given the WAN delays and the real-time nature of the workload. Despite quite a bit of existing work [41], we know of no cut-and-dried consistency model that would fit this scenario perfectly, and it is not immediately clear (to us) what one should look like. The answer may depend partly on the scenario – a concert may need weaker consistency than a basketball game – and even on the situation: if Alice shoots a paintball at Bob, she needs strong consistency *at that moment* to decide whether Bob has been hit, even though weaker consistency may be fine otherwise.
Key challenges: Due to the heterogeneous workload, picking a good consistency model for the metaverse is much harder than for a specific game – there may not even be a single "right" answer at all! Maintaining it efficiently *at scale* is another challenge; there are approaches that can avoid coordination some of the time [5], which could be useful.

3.10 Efficiency

Running virtual worlds on a shared, cloud-like infrastructure could enable a number of interesting optimizations. One example is vectorization: suppose a creator has made a popular "house" template that several experiences are using in cities. In this case, some of the underlying computation is identical and could potentially be vectorized, somewhat analogous to Orochi [71], or the parallel simulations in Madrona [64]. Similarly, it may be possible to apply a form of deduplication [27]: if the same script or object exists in several different instances, one could save space by storing it only once.

Of course, the state of identical objects may diverge over time; for instance, a user may choose to paint her walls in a different color, or someone might smash a window. In a pure copy-on-write solution, such a divergence would end the benefit from deduplication. But we hypothesize that there is a way to create the equivalent of union filesystems, where a single, immutable filesystem can be customized with a mutable delta layer and the user-visible state is the union of the two. In other words, a scalable platform could use

vectorization and deduplication for the first layer and then apply small per-instance customizations on top of the result. **Key challenges:** A metaverse simulation will likely represent a giant computational workload, so it is important to increase efficiency, in order to keep the service affordable.

3.11 Scheduling

Unlike many existing cloud workloads, metaverse computations are highly time-sensitive, so careful scheduling is critical. However, they are also highly variable: players and objects can move around, scripts can run at irregular times, and the number and type of objects can change dramatically – for instance, an explosion can turn a large building into rubble. This creates a challenging scheduling problem.

So far, the scheduling literature does not seem to cover this scenario very well: existing cluster scheduling techniques, like RackSched [84] or FIRM [57], can support latency-sensitive services but do not offer real-time guarantees, while existing real-time scheduling techniques tend to focus on small-scale and/or embedded systems and do not scale to data centers. However, there are techniques that could be adapted or extended: for instance, compositional scheduling [65] can scale up more traditional real-time scheduling, and multi-mode techniques [58] can be used to adjust schedules while preserving the timing guarantees during transitions.

However, a metaverse workload also provides interesting new opportunities for scheduling. For instance, it is highly flexible: although a scheduler cannot necessarily change the deadline for the next frame, it can change the level of detail for certain objects, run physics with larger time steps, approximate computations or run them with lower precision, or even temporarily drop some functions (such as shadows) entirely. Techniques such as mixed-criticality scheduling [74] could exploit this by prioritizing critical tasks during overload. It is also *somewhat* predictable at short timescales: if many players start to congregate in a small area, the scheduler can migrate parts of this area to other nodes. Notice that this workload is more transparent to the infrastructure than a traditional compute workload: an EC2 server cannot easily predict when the needs of a VM are about to change, but in this case, a lot of semantic information (such as player positions and trajectories) is visible to the platform.
Key challenges: The scheduling requirements of the metaverse are quite different from almost everything else we have seen in the literature: it involves a unique combination of massive scale, highly variability, stringent timing requirements, and flexibility in several dimensions. This is both a challenge and an interesting opportunity.

3.12 Security

If the simulation runs entirely on infrastructure servers, it is relatively easy to guarantee, e.g., that players cannot walk through walls or open doors without a key. But once some of the computation is moved to clients, there is a risk of cheating: some users could hack their devices and make changes to game state that their device controls. Since there are compelling reasons for this move (see Section 3.3), the question is how one could guarantee integrity in this case.

One way would be to use traditional anti-cheat software on the client side [32, 55, 62]. This is generally effective, but a) it is fairly specific to known forms of cheating, which leads to a persistent cat-and-mouse game that the platform operators cannot hope to win, and b) there are sophisticated attacks that this approach cannot detect. Another, more general approach would be auditing: the infrastructure has access to most of the information the client is using, so it can repeat, say, a small fraction of the computations and compare the results to the ones the client has sent, analogous to [11, 29]. This could also help, e.g., with hardware malfunctions [31].

The question is *which* computations the infrastructure should audit. Random choices would reliably catch continuous misbehavior that affects lots of computations (e.g., if the client has changed the laws of physics or made an object disappear entirely), but a smart cheater might make changes only when it matters most – say, disable collision detection only for the one bullet that would have hit them. Thus, it would be useful to focus audits on "critical" computations – perhaps with some help from the creators, who could annotate important pieces of game state, such as player health or the number of gold coins. If a problem is detected on a player device, the infrastructure could simply take over its part of the simulation, analogous to [1].
Key challenges: A metaverse platform would provide very clear incentives for attacks in combination with a large attack surface, since much of the "platform" consists of client devices. This problem has been studied, e.g., in the context of peer-to-peer games [80], but a metaverse service would have an infrastructure component as well, and this should help [1]. In addition, many of the "classic" security challenges (user authentication, confidentiality, etc.) apply as well.

4 DISCUSSION

Scaling the metaverse strikes us as a prime example of a problem that can only be solved collaboratively and by working across domains – including at least graphics (SIGGRAPH), gaming (GDC), systems (SoCC/SOSP/OSDI), and real-time (RTSS/RTAS). Many of the challenges – rendering, animation, etc. – lie squarely in the traditional graphics domain, but research in that domain so far has not focused on massive scalability or hard real-time guarantees. The systems community has developed many massive-scale systems over the years, but it has been focusing mostly on asynchronous systems, without much attention to timing [81]. Timing is a key focus of the real-time community, but that community,

once again, has not been focusing much on either scalability or virtual worlds. And the (typical) systems and real-time researchers do not have the necessary expertise to structure the "gameplay" for the metaverse, so that the worlds are fun for the players and keep them engaged. Clearly, each community has an important piece of the puzzle, but so far, there is very little work that brings all of the pieces together.

Unfortunately, it also does not seem likely that the key technologies could be developed independently and then simply combined. Virtual-world workloads are quite different from the traditional workloads of the systems community – for instance, due to their timing requirements and their pervasive dependencies – and the key to scaling them may lie in their specific properties, such as their spatial structure. Likewise, the metaverse's scheduling problems are unlike any we have seen in the existing real-time literature – for instance, due to their unique requirements, their scale, and their many degrees of freedom – and it seems unlikely that an existing technique, say, from embedded systems, would apply out of the box. We suspect that a truly practical solution can only be developed *together* by all four communities.

5 RELATED WORK

The metaverse vision has received a lot of attention, both from the research community and from the press, so there is a lot of existing work; see, e.g., [78] for a recent survey. However, much of this work focuses on AR/VR headsets, computer vision, etc.; here, we focus specifically on systems challenges and especially on scalability.

To our knowledge, there is very little published research on scaling virtual worlds to thousands of players and beyond. The closest instance we know of is a concert by musical artist Aurora within the mobile game Sky: Children of the Light, which was watched by more than 1.6 million people [70], with 4,000 players in each instance of the concert. This was a hand-tailored system that took a lot of careful engineering. Other platforms seem to be limited to a few hundred players as well; for instance, van der Sar et al. [73] studied Minecraft and found a limit of about 300–400 players.

Donkervliet et al. [19] suggests using serverless computing to distribute the game server across multiple machines and to provide it as a cloud service. While this would allow games to grow beyond the capacity of a single server, it would not lead to efficient scalability, at least not without the locality properties we sketch here. Turchini et al. [72] sketches a similar approach but envisions a P2P architecture, with cloud VMs helping with crowded zones when necessary. JOT [51] breaks a monolithic game server into modules but does so to make programming easier; it does not mention scalability.

Improbable [34] has probably come closest to achieving our vision. Its technology is impressive, but so far it has had limited success, reportedly at least in part due to high cost and a difficult programming model [66]. Its latest platform supports at least 4,500 players per instance [68]. Hadean [28] has apparently gone further during internal testing [69].

A variety of frameworks for scalable distributed systems have been applied to online games, but, to our knowledge, none of them can provide a single *shared* virtual world for a large number of players. For instance, Akka [2] is used by Fortnite under the hood [76], and Microsoft Orleans [47] was used to provide cloud services for Halo [85], but Halo's Big Team Battle mode only supports up to 24 players [30], and Fortnite's Battle Royale mode is limited to 100 players [79]. Some research prototypes, such as RTF [23], also exist. Vircadia [75] and Overte [50] are open-source metaverse platforms, but their scalability seems comparable to that of the other platforms we mentioned; for instance, Vircadia's home page says that its virtual worlds "can scale to hold hundreds of people in the same space without instancing".

Cloud gaming is a related approach in which cloud servers perform both game execution and rendering on behalf of the clients; the client devices merely display output frames and accept inputs. This approach involves some related challenges, e.g., a slightly different form of speculation [40, 77]. Due to the high latencies and limited bandwidth in a WAN environment, it is generally difficult to match the user experience of a local game with this approach.

6 CONCLUSION

Building a large-scale metaverse experience is challenging: this seems clear from the limitations of existing platforms, none of which can currently handle more than a few hundred interacting players. Some of the challenges may be nontechnical, such as difficult programming models and high cost. However, our central hypothesis in this paper is that there are hard technical challenges at the heart of all this, many of which have yet to be solved convincingly.

We think that these challenges could be fertile ground for the cloud community. True, the workload will be unfamiliar to many of us: some of the technology originated in multiplayer gaming, and so far the problem has not had much visibility here. But many of the solutions seem squarely within our expertise: scalability, consistency, scheduling, and load balancing are all classical cloud questions. This new class of applications could be an exciting opportunity, and solving the questions we have sketched here could be the key to making the metaverse a reality.

ACKNOWLEDGMENTS

We thank our shepherd Luo Mai and the anonymous reviewers for their thoughtful feedback. This vision is the result of discussions with many others at Roblox, including Alan Jeffrey, Arjun Guha, Cem Yuksel, Layla Mah, Mitesh Shah, Sheldon Andrews, Tania Lorido-Botrán, and Victor Zordan.

REFERENCES

[1] P. Aditya, M. Zhao, Y. Lin, A. Haeberlen, P. Druschel, B. Maggs, and B. Wishon. Reliable client accounting for hybrid content-distribution networks. In *Proc. NSDI*, Aug. 2012.
[2] Akka. https://akka.io/.
[3] H. Al Maruf and M. Chowdhury. Effectively prefetching remote memory with Leap. In *Proc. ATC*, 2020.
[4] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *Proc. POPL*, 2011.
[5] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *Proc. VLDB Endow.*, 8(3):185–196, 2014.
[6] K. Barandy. What will Mark Zuckerberg's metaverse actually look like, and who is asking for it? Designboom, https://www.designboom.com/technology/mark-zuckerberg-metaverse-horizon-worlds-avatar-virtual-reality-08-26-2022.
[7] G. Barthe, B. Köpf, F. Olmedo, and S. Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In *Proc. POPL*, 2012.
[8] B. Beckman, F. Wieland, L. Blume, M. DiLoreto, P. Hontalas, P. Larouche, K. Sturdevant, J. Tupman, V. Varren, J. Wedel, H. Younger, and S. Bellenot. Distributed simulation and the Time Warp operating system. In *Proc. SOSP*, 1987.
[9] A. Bharambe, J. R. Douceur, J. R. Lorch, T. Moscibroda, J. Pang, S. Seshan, and X. Zhuang. Donnybrook: Enabling large-scale, high-speed, peer-to-peer games. In *Proc. SIGCOMM*, 2008.
[10] P. Buneman, S. Khanna, and W.-C. Tan. Why and Where: A characterization of data provenance. In *Proc. ICDT*, Jan. 2001.
[11] C. Chambers, W. Feng, W. Feng, and D. Saha. Mitigating information exposure to cheaters in real-time strategy games. In *Proc. NOSSDAV*, June 2005.
[12] K. Channa. Max player count increased (in Beta. Aug. 2020. https://devforum.roblox.com/t/max-player-count-increased-in-beta/722045.
[13] J. Chen, B. Wu, M. Delap, B. Knutsson, H. Lu, and C. Amza. Locality aware dynamic load management for massively multiplayer games. In *Proc. PPoPP*, 2005.
[14] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. NSDI*, 2005.
[15] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proc. SOSP*, 2013.
[16] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proc. NSDI*, Apr. 2008.
[17] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI*, 2004.
[18] D. J. DeWitt and M. Stonebraker. MapReduce: A major step backwards, Jan. 2008. Archived at https://homes.cs.washington.edu/~billhowe/mapreduce_a_major_step_backwards.html.
[19] J. Donkervliet, A. Trivedi, and A. Iosup. Towards supporting millions of users in modifiable virtual environments by redesigning minecraft-like games as serverless systems. In *Proc. HotCloud*, 2020.
[20] G. W. Dunlap, D. Lucchetti, P. M. Chen, and M. Fetterman. Execution replay for multiprocessor virtual machines. In *Proc. CEE*, Mar. 2008.
[21] T. Ford. Overwatch gameplay architecture and netcode. Talk at GDC 2017, available at https://www.youtube.com/watch?v=W3aieHjyNvw.
[22] M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce. Linear dependent types for differential privacy. In *Proc. POPL*, 2013.
[23] F. Glinka, A. Ploss, S. Gorlatch, and J. Müller-Iden. High-level development of multiserver online games. *Int. J. Comput. Games Technol.*, Jan. 2008.
[24] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proc. OSDI*, 2012.
[25] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.
[26] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. SIGMOD*, 1993.
[27] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. In *Proc. OSDI*, 2008.
[28] Hadean. https://hadean.com/.
[29] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel. Accountable virtual machines. In *Proc. OSDI*, Oct. 2010.
[30] Halo Support. Guide to Halo Infinite multiplayer game mode rules. https://support.halowaypoint.com/hc/en-us/articles/4407649063316-Guide-to-Halo-Infinite-multiplayer-game-mode-rules.
[31] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat. Cores that don't count. In *Proc. HotOS*, 2021.
[32] G. Hoglund. 4.5 million copies of EULA-compliant spyware. http://www.rootkit.com/blog.php?newsid=358.
[33] G. Hoglund and G. McGraw. *Exploiting Online Games: Cheating Massively Distributed Systems.* Addison-Wesley, 2007.
[34] Improbable. https://www.improbable.io/.
[35] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. Diloreto. Time warp operating system. In *Proc. SOSP*, 1987.
[36] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: Language support for building distributed systems. In *Proc. PLDI*, 2007.
[37] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *Proc. INFOCOM*, 2004.
[38] S. Kumar, D. E. Culler, and R. A. Popa. MAGE: Nearly zero-cost virtual memory for secure computation. In *Proc. OSDI*, 2021.
[39] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4):471–482, 1987.
[40] K. Lee, D. Chu, E. Cuervo, J. Kopf, Y. Degtyarev, S. Grizan, A. Wolman, and J. Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proc. MobiSys*, 2015.
[41] H. Liu, M. Bowman, and F. Chang. Survey of state melding in virtual worlds. *ACM Comput. Surv.*, 44(4), Sept. 2012.
[42] S. Liu, X. Xu, and M. Claypool. A survey and taxonomy of latency compensation techniques for network computer games. *ACM Computing Surveys*, 54(11s), Sept. 2022.
[43] B. T. Loo, T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. *Commun. ACM*, 52(11):87–95, 2009.
[44] M. McGuire. The network is the Renderer, 2022. Keynote at MMSys'22.
[45] M. McGuire and O. C. Jenkins. *Creating Games.* A K Peters/CRC Press, 2008.
[46] F. McSherry. Privacy integrated queries. In *Proc. SIGMOD*, June 2009.
[47] Microsoft Orleans. https://learn.microsoft.com/en-us/dotnet/orleans/.
[48] B. Mirtich. Timewarp rigid body simulation. In *Proc. SIGGRAPH*, 2000.
[49] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *Proc. SOSP*, 2005.
[50] Overte. https://overte.org/.
[51] G. Paiva Amador and A. Gomes. JOT: A modular multi-purpose minimalistic massively multiplayer online game engine. In *Proc. Videojogos*,

2016.

[52] L. Pantel and L. C. Wolf. On the suitability of dead reckoning schemes for games. In *Proc. NetGames*, 2002.

[53] A. Pesce. Rendering the Metaverse across space and time, 2020. Digital Dragons; http://c0de517e.blogspot.com/2020/12/digital-dragons-2020.html.

[54] R. S. Peterson and E. G. Sirer. Antfarm: Efficient content distribution with managed swarms. In *Proc. NSDI*, 2009.

[55] PunkBuster web site. http://www.evenbalance.com/.

[56] R. Pusch. Explaining how fighting games use delay-based and rollback netcode, Oct. 2019. Ars Technica, https://arstechnica.com/gaming/2019/10/explaining-how-fighting-games-use-delay-based-and-rollback-netcode/.

[57] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer. FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices. In *Proc. OSDI*, 2020.

[58] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26:161–197, 2004.

[59] Roblox. Luau language homepage. https://luau-lang.org/.

[60] Roblox. Monetization. https://create.roblox.com/docs/production/monetization.

[61] Roblox. Roblox reports March 2023 key metrics. https://ir.roblox.com/news/news-details/2023/Roblox-Reports-March-2023-Key-Metrics/default.aspx.

[62] Roblox Developer Blug. Welcoming Byfron to Roblox, Oct. 2022. https://devforum.roblox.com/t/welcoming-byfron-to-roblox/2018233.

[63] R. Rodrigues and P. Druschel. Peer-to-peer systems. *Commun. ACM*, 53(10):72–82, Oct. 2010.

[64] B. Shacklett, L. G. Rosenzweig, Z. Xie, B. Sarkar, A. Szot, E. Wijmans, V. Koltun, D. Batra, and K. Fatahalian. An extensible, data-oriented architecture for high-performance, many-world simulation. *ACM Trans. Graph.*, 42(4), July 2023.

[65] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proc. RTSS*, 2003.

[66] K. Shubber. Improbable Worlds' dream of revolutionising gaming is fading. *Financial Times*, Mar. 2021.

[67] N. Stephenson. *Snow Crash*. Bantam, 1992.

[68] D. Takahashi. Herman Narula: How Improbable put 4,500 bored apes in the same metaverse space. VentureBeat, August 3, 2022; https://venturebeat.com/games/herman-narula-how-improbable-put-4500-bored-apes-in-the-same- metaverse-space/.

[69] D. Takahashi. Hadean raises $30m to create infrastructure for the metaverse, Sept. 2022. https://venturebeat.com/games/hadean-raises-30m-to-create-infrastructure-for-the-metaverse/.

[70] D. Takahashi. How 1.6M people watched a concert in Sky: Children of the Light, Dec. 2022.

[71] C. Tan, L. Yu, J. B. Leners, and M. Walfish. The efficient server audit problem, deduplicated re-execution, and the Web. In *Proc. SOSP*, 2017.

[72] G. Turchini, S. Monnet, and O. Marin. Scalability and availability for massively multiplayer online games. In *Proc. EDCC*, Sept. 2015.

[73] J. van der Sar, J. Donkervliet, and A. Iosup. Yardstick: A benchmark for Minecraft-like services. In *Proc. ICPE*, 2019.

[74] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proc. RTSS*, 2007.

[75] Vircadia. https://vircadia.com/.

[76] A. Woodie. Inside Fortnite's massive data analytics pipeline, July 2018. Datanami, July 31, 2018; https://www.datanami.com/2018/07/31/inside-fortnites-massive-data-analytics-pipeline/.

[77] J. Wu, Y. Guan, Q. Mao, Y. Cui, Z. Guo, and X. Zhang. ZGaming: Zero-latency 3D cloud gaming by image prediction. In *Proc. SIGCOMM*, 2023.

[78] M. Xu, W. C. Ng, W. Y. B. Lim, J. Kang, Z. Xiong, D. Niyato, Q. Yang, X. S. Shen, and C. Miao. A full dive into realizing the edge-enabled metaverse: Visions, enabling technologies, and challenges. *IEEE Commun. Surveys & Tutorials*, 2022.

[79] J. Yaden. Fortnite cannot support more than 100 players, Epic says. Gamespot, March 30, 2023; https://www.gamespot.com/articles/fortnite-cannot-support-more-than-100-players-epic-says/1100-6512850/.

[80] A. Yahyavi and B. Kemme. Peer-to-peer architectures for massively multiplayer online games: A survey. *ACM Comput. Surv.*, 46(1), July 2013.

[81] T. Yang, R. Gifford, A. Haeberlen, and L. T. X. Phan. The synchronous data center. In *Proc. HotOS*, May 2019.

[82] M. Zhang, Y. Wu, K. Chen, X. Qian, X. Li, and W. Zheng. Exploring the hidden dimension in graph processing. In *Proc. OSDI*, 2016.

[83] M. Zhao, P. Aditya, A. Chen, Y. Lin, A. Haeberlen, P. Druschel, B. Maggs, B. Wishon, and M. Ponec. Peer-assisted content distribution in Akamai NetSession. In *Proc. IMC*, 2013.

[84] H. Zhu, K. Kaffes, Z. Chen, Z. Liu, C. Kozyrakis, I. Stoica, and X. Jin. RackSched: A Microsecond-Scale scheduler for Rack-Scale computers. In *Proc. OSDI*, 2020.

[85] R. V. Zicari. Orleans, the technology behind Xbox Halo4 and Halo5. interview with Phil Bernstein. ODBMS Industry Watch, February 9, 2016; https://www.odbms.org/blog/2016/02/orleans-the-technology-behind-xbox-halo4-and-halo5-interview- with-phil-bernstein/.