A Scalable Server for 3D Metaverses

http://sirikata.com

Ewen Cheslack-Postava[1], Tahir Azim[1], Behram F.T. Mistree[1],
Daniel Reiter Horn[1], Jeff Terrace[2], Philip Levis[1], and Michael J. Freedman[2]

[1]Stanford University [2]Princeton University

Abstract

Metaverses are three-dimensional virtual worlds where anyone can add and script new objects. Metaverses today, such as Second Life, are dull, lifeless, and stagnant because users can see and interact with only a tiny region around them, rather than a large and immersive world. Current metaverses impose this distance restriction on visibility and interaction in order to scale to large worlds, as the restriction avoids appreciable shared state in underlying distributed systems.

We present the design and implementation of the Sirikata metaverse server. The Sirikata server scales to support large, complex worlds, even as it allows users to see and interact with the entire world. It achieves both goals simultaneously by leveraging properties of the real world and 3D environments in its core systems, such as a novel distributed data structure for virtual object queries based on visible size. We evaluate core services in isolation as well as part of the entire system, demonstrating that these novel designs do not sacrifice performance. Applications developed by Sirikata users support our claim that removing the distance restriction enables new, compelling applications that are infeasible in today's metaverses.

1. INTRODUCTION

Virtual worlds are three-dimensional graphical environments where people can interact, engage in games, and collaborate. This paper focuses on one particular class of virtual worlds: "metaverses," such as Second Life, where anyone can add new objects to the world by creating 3D models and writing scripts to control them.

Users of today's metaverses miss out on a truly immersive experience because they can see and interact with only the tiny, local region around them. For example, Second Life rendered completely is a fantastic and eye-popping megalopolis, a fictional Shanghai. But users are limited to their local surroundings because Second Life only displays nearby objects and imposes a maximum object interaction distance of ~100 meters. Imposing a distance restriction allows a metaverse to scale easily because a server only shares state and communicates with the servers that manage neighboring regions. But this scalability comes at a cost of user experience. Today's metaverses are lonely and empty [26] and users cannot see what awaits them beyond their interaction range. A distant building in a virtual city should be able to attract a user's attention, and she should be able to select it and send it messages to learn more about it. What is it? Who owns it? Such a simple interaction is impossible in today's metaverses.

In contrast to metaverses, "games," such as World of Warcraft, avoid this problem because the game provider is the centralized author of all world content and the system is application-specific. Central control of content allows the provider to design content around technical limitations. For example, games can precompute scene optimizations (e.g., binary space partition trees for visibility calcuations [31] and imposters for efficient rendering [16]) because game worlds are mostly static. As narrow, tailored applications, games can leverage game mechanics for improved performance: Donnybrook, for instance, showed how focusing updates around the flag bearer in capture the flag leads to lower bandwidth demand [5]. Application-specific optimizations commonly permeate the designs of these systems. For example, World of Warcraft scales to millions of users by dividing users across hundreds of "realms," or copies of the world, and most game content occurs within "instances" where a group interacts with the game privately.

Metaverses differ greatly from games and encounter correspondingly different technical challenges. Instead of static, centrally authored content serving a single application, a metaverse presents a large continous world populated by many user-generated applications whose dynamic behavior, placement, and appearance are controlled by their owners. The tricks used in games are not sufficient in a metaverse: a user could add a skyscraper using a new mesh with unknown behavior, which must quickly be made visible to users and be able to interact with other objects and avatars.

We believe that metaverse users should be able to view and interact with the entire world. However, to support a world with billions of dynamic user-generated and user-controlled objects, metaverses must somehow limit interaction. Without limits the system would quickly exceed capacity: clients cannot render the entire world in full detail, position updates for every object would saturate a server's outgoing capacity, and even sending a list of all objects to a client would overload a server.

The research contribution of this paper is establishing a novel principle for how to build scalable systems for dynamic, user-extensible virtual environments. Scaling these systems is challenging because the common approach in systems of share-nothing partitionings leads to the very problems we observe in today's metaverses, restricting the view of the world and scope of interaction. Simply removing these limitations leads to unscalable systems. For example, displaying the entire world requires tracking distant objects and therefore requires global queries for objects. Without additional constraints, global queries would require all pairs of servers to exchange queries, limiting the world to just a few servers.

The graphics research community has not tackled this problem in a significant way, typically focusing their systems only on a subset of the requirements of metaverses. For example, render farms are distributed systems that handle large datasets but work offline, and real-time rendering is commonly applied to data that fit in memory on a single host and permit precomputing acceleration data structures. Many ideas from graphics research can be borrowed, but different decisions must be made to work in real-time, in a distributed system, and handle dynamic user-generated content.

This paper presents the design and implementation of a metaverse server that allows clients to see and interact with a large and complex world, even as the system grows to many servers. The key insight for achieving this scalability goal is that a virtual world, being a 3D, geometric environment, has many similarities to the real world. By leveraging real-world-like constraints, the server can simultaneously scale to a large world and provide a natural as well as intuitive experience.

Consider the server's object discovery service, which informs clients about objects in the world. Rather than return the closest objects (which restricts visibility to a small area) or all of the world's billions of objects (which does not scale), the Sirikata server returns objects that have the largest *solid angle*. Solid angle is roughly equivalent to how many pixels an object occupies on screen, so Sirikata returns only objects which could be seen by the user on screen. While this query is global (an avatar can see a building that is many servers away), the number of distant objects and associated network traffic is small. Unlike a distance metric, solid angles neither hide distant, visually important buildings nor do they cause them to suddenly and jarringly pop into existence.

This principle of leveraging real-world constraints underlies all of the server's core systems and is critical to scalability. The server is part of the open-source Sirikata platform [33], already used in applications ranging from virtual museums [3] to social spaces for cancer patients [19]. The paper describes four core components, focusing on how each can scale by applying this principle:

- A server partitioning service that divides a world into regions and maps regions to servers. Assuming an object density distribution similar to the real world leads to a design using a distributed, split-axis kd-tree, with a highly replicated, stable upper tree and distributed lower trees to spread load.
- An object discovery service that prioritizes objects by visual importance, choosing objects with larger solid angles, with a novel distributed data structure and supporting query processor.
- A message routing table that maps object identifiers to servers. Every server can forward to any object in the world, but the routing table scales well because, like the real world, most interactions are local and most objects are stationary.
- A message forwarder that guarantees a minimum throughput even as the number of communicating objects grows very large, decaying smoothly over distance.

We focus on the server partitioning and object discovery services, providing a brief overview of the routing table and forwarder for completeness. We evaluate the first two services in isolation as well as the entire Sirikata server, finding it can scale to large, visually rich worlds with many applications. We also report on our experiences with users developing applications in Sirikata, including some which are difficult or impossible in existing metaverses due to their use of a distance limitation. Finally, while this paper focuses on metaverses, we believe its principles can be applied more generally to systems that bridge physical and virtual environments, such as augmented reality and ubiquitous computing.

2. METAVERSES TODAY: SECOND LIFE

A metaverse is a virtual world that users can extend by adding objects and scripting them to provide interesting applications. This user-extensibility contrasts with games, where all content originates from a central author. A metaverse provides a few basic services with which to build applications:

- storing the location of objects and a reference to the graphical models of objects,
- updating location and model state based on script commands and physics simulation,

- executing object scripts and generating script events, such as timers and inter-object communication,
- telling objects and clients what objects they can see and interact with.

Second Life, today's most popular metaverse, supports over 1 million monthly active users [37]. Its design is representative of how current metaverses decompose internal systems and support applications. Second Life statically partitions land into 256m × 256m regions called "sims," each managed by a different server. A sim's server is responsible for all activity within the sim, including the above services.

Second Life uses distance to control object discovery and inter-object communication. Servers prioritize objects for clients based on multiple factors, but generally clients cannot see objects farther than 100 meters away. Scripted objects (applications) cannot detect objects farther than 15 meters away. The basic communication primitive is a geometric broadcast to all objects within 10, 20, or 100 meters. Second Life also provides a very limited form of unicast which is unsuitable for interactive applications due to its low throughput and high latency.

These restrictions are critical for Second Life to be able scale to its current size. They allow the simulation to be nearly shared nothing: servers only coordinate with their geometric neighbors. Because the region-to-server mapping is static in Second Life, the system cannot respond to variations in load and most sims can support only a small number (40) of avatars. As a result, most of the world lies empty [26].

Relaxing these restrictions may enable virtual worlds to move closer to their imagined potential. Users will be able to view sweeping vistas, and the applications they interact with will be able to span the world. Users will be able to serendipitously discover events throughout the world; players in a game will be able to monitor and aid distant allies; and real-world data will take on new meaning when overlaid onto a virtual replica of Earth. Enabling this vision, however, requires novel distributed systems principles and techniques.

3. SIRIKATA SERVERS

Sirikata servers allow users to see and interact with a large virtual world that runs distributed over many hosts. It breaks the tension between scale and scope by leveraging constraints and restrictions similar to the real world throughout its systems. This section describes the internals of a Sirikata server, as well as its role as one part of a complete metaverse system.

3.1 Sirikata Applications and Overview

Sirikata has a very different application model than Second Life because it allows interaction with distant objects. Sirikata's model greatly resembles the web of today: application objects send code and data to user objects. For example, when an avatar activates a chess board, the board sends the avatar a UI, which exchanges application messages with the board object. This application model makes long-range inter-object communication a key system service. Applications are written in the Emerson language, a JavaScript dialect we developed that introduces mechanisms to address many of the security problems web applications have exposed [25].

Sirikata splits the implementation of a virtual world into three components. Object hosts run the object scripts that implement user-defined applications. Scripted objects populate and move through the world, sending and receiving application-level messages. A content distribution network (CDN) stores and delivers large pieces of data, such as graphical models and textures. Space servers (or just "servers"), the subject of this paper, are responsible for the virtual world itself. They are authoritative on the presence and positions of objects in the world, simulate physics, inform observers of other relevant objects and keep them up-to-date on those objects' positions. Finally, servers are responsible for routing the application-level messages between scripted objects.

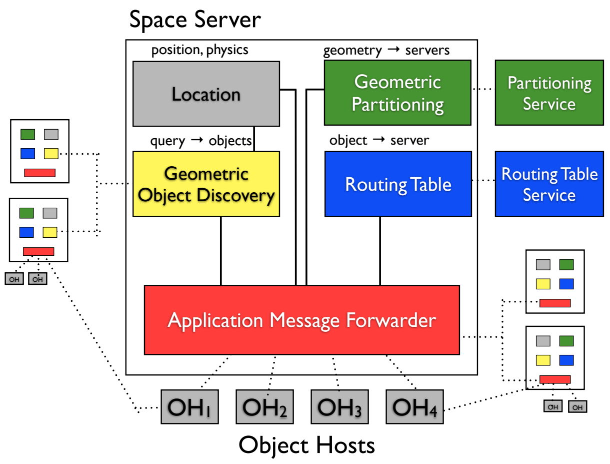
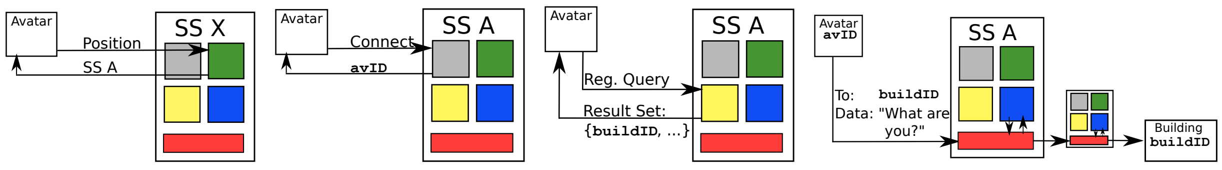


Figure 1: Sirikata server internals

3.2 Server Internals

Figure 1 shows the interaction of object hosts with servers and the internal decomposition of a Sirikata server into its major services. Each server is responsible for a subset of the geometric area of the world. Object hosts connect to the servers which manage the regions occupied by their objects. As shown, a single object host may connect to many servers, and many object hosts may connect to each server. Eliding some ancillary services, such as login/authentication, for brevity, the major services are as follows:



(a) Server discovery (b) Enter world (c) Discover building (d) Message building

Figure 2: Avatar connection and messaging example. Service positions and colors are consistent with Figure 1.

- The **geometric server partitioning** service maintains a distributed data structure for controlling and querying the partitioning of the world across servers (Section 4).
- The **location table** stores the position, orientation, and motion of objects, which the physics engine (based on Bullet [12]) updates. Because this is a traditional component and not novel, this paper does not discuss it in depth.
- The **geometric object discovery** service handles queries to discover other objects, streaming back object identifiers and subscribing the querier for location updates (Section 5).
- The **application message routing** table maintains a durable, consistent, distributed mapping from an object identifier to the space server responsible for it (Section 6.1).
- The **application message forwarder** forwards inter-object (application) messages, using the routing table to determine the next hop. It responds to congestion by giving greater weight to traffic between closer and larger objects (Section 6.2).

Each of these services leverages some property of the physical world in order to scale. For example, the geometric partitioning service leverages the fact that most objects in the world are stationary, such that most changes to its data structure are small and local. As a second example, the message router weights inter-object message flows similarly to the inverse-square falloff of electromagnetic radiation intensity over distance. This weighting provides a natural and intuitive experience: larger and closer objects have greater communication capacity than small and distant ones. Furthermore, the sum of weights for objects in a fixed region converges to a constant, so Sirikata can guarantee that nearby objects cannot be drowned out.

3.3 Example Execution

To explain how Sirikata's services create a working virtual world system, we use the example from Section 1 of a user logging into a world, seeing an interesting building, and selecting it to learn more about it. Figure 2 illustrates what happens.

To log in, the object host running the user's avatar connects to any server and requests an initial position. The space server, SS X, looks up this location with the partitioning service and redirects the avatar to the server managing that region (Figure 2(a)). The avatar connects to and authenticates with the new server, SS A. The server enters the avatar state into the location table and adds entries in the routing table and query processor (Figure 2(b)). To display the world, the avatar registers an object discovery query (Figure 2(c)). The discovery service streams object identifiers and important data, such as position and a URL for each object's graphical model. One such object is the distant, interesting building with object identifier `buildID`.

Figure 2(d) shows how the avatar sends an application-level message to the building object when the user selects it. The message takes three hops. First, the avatar's object host sends it to the server. The server consults its routing table to determine which server is responsible for the building and passes the message to the forwarder. Finally, the destination server forwards it to the building's object host. The building sends a message in reply, containing a web page of information, which an in-world web browser displays.

The next four sections follow the steps of this example and describe the design and implementation of each of the server's core services.

4. SERVER PARTITIONING

As shown in Figure 2(a), when an avatar joins the world, it queries a known server's partitioning service to determine which space server is currently authoritative for its future position. This is necessary because, to support changes in load, Sirikata's mapping of geometric regions to space servers is dynamic.

The geometric server partitioning service manages this mapping with two abstractions. Given a bounding region, it identifies the space servers that manage the region, and given a space server identifier, it returns the bounding region managed by that server. It also chooses when to merge and split regions to balance load across servers, notifying the servers that they should change the region they manage and migrate objects.

The key challenge in implementing the partitioning ser-

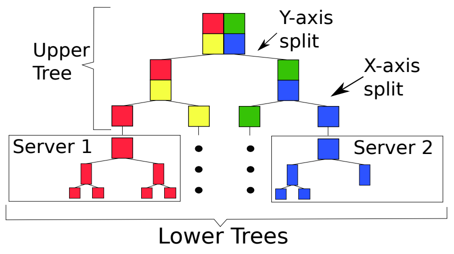


Figure 3: The kd-tree used by the partitioning service and its division into upper and lower trees.

vice is designing a scalable, distributed space-partitioning data structure to efficiently answer bounding-box queries. This data structure should support Earth-scale worlds, make splitting and merging regions for load balancing efficient and simple, and not require major restructuring as objects move, appear, and disappear.

The partitioning service uses a split-axis kd-tree as its basic data structure. Well-known to the graphics community, kd-trees are binary trees that recursively split a region using axis-aligned splitting planes [4]. The partitioning service uses a *split-axis* kd-tree, which always splits regions in equal halves, using the same axis for all splits at a given level of the tree. The split-axis kd-tree cycles between the x- and y- axes to select the splitting planes at successive levels of the tree. Splitting continues as long as the resulting regions (the leaves of the tree) contain more than n objects. Figure 3 shows an example of this process. Split-axis kd-trees are simpler to build and maintain than ones which optimize the placement of split-planes because the plane is not affected by object motion or object distribution. The drawback is that they are deeper than optimally-constructed kd-trees, which reduces their performance for spatial queries.

An experiment quantifies this tradeoff. Using historical and projected real-world population data from 1990–2015 that partitions Earth into ~29 million cells [10], we construct a split-axis kd-tree with at most 40 people per leaf node. The maximum depth of this kd-tree is never more than 50% larger than the optimal kd-tree and the average depth increases by only 20%. As performance evaluations in Section 7.2 later show, this additional tree traversal time is negligible compared to a query's network latency.

4.1 Distributing the Partitioning Service

Although the entire kd-tree of an Earth-sized world can fit in-memory on a single server, the partitioning service must distribute the kd-tree across many hosts for availability, fault-tolerance, and to support high query and update rates. A straw man approach with strongly-consistent replicas of the full kd-tree becomes too costly as regions split and merge for load balancing. The tree has 29 million leaves, and avatar movement can cause frequent splits and merges: large movements of avatars could easily overwhelm a single data structure's ability to process strongly consistent updates.

Depth:	10	12	14	16	18	20
World Pop	6.0	40.0	238.0	966.0	4642.0	22,140.0
Zipf	0.0	0.0	0.0	0.0	1240.4	290,000.0

Table 1: Number of changes in the partitioning service's upper tree with different choices of cut depth.

Instead, the partitioning service divides the kd-tree into two parts at a fixed cut depth (Figure 3): an upper tree and a set of lower trees. The upper tree is replicated across all partitioning servers. The lower trees are managed by individual partitioning servers. Each upper tree leaf specifies the domain name of the partitioning server that stores the lower tree rooted at that leaf. Any partitioning service node can answer bounding box queries by traversing the replicated upper tree and forwarding the query to servers managing the relevant lower trees.

This design is effective because the split-axis kd-tree makes the upper tree very stable. Table 1 demonstrates upper tree stability, showing the number of changes in the upper tree for different upper tree depths. Two distributions are tested. The first uses the world population data discussed in this section to count the number of changes in the upper tree as world population increased from 1990–2015. The second is Zipfian, a distribution observed in multiplayer games [6], and uses the same world population cells, assigning each a density of $\frac{D}{i}$ randomly, where $D = 120,000\frac{objects}{km^2}$ and $i = 1, 2, \ldots$ The table shows changes averaged over ten different Zipf distributions to demonstrate the stability of the upper tree as the densest parts of the world shift around. A sufficiently high cut mostly avoids changes to the upper tree: at a depth of 10 there no changes for the Zipf distribution and only 6 changes over 25 years for the world population data. Even at a depth of 16 there are fewer than 1000 changes — less than 2% of nodes at or above that depth. This low write rate makes it feasible to replicate the upper tree across all partitioning servers.

4.2 Load Balancing

The partitioning service dynamically splits and merges regions to balance server load. It currently uses an ad-hoc strategy: split a region in two when the number of objects inside it exceeds a threshold, and merge two sibling regions if both have fewer objects than $\frac{1}{4}$ the threshold. This approach can create a logarithmic number of lightly-loaded servers if a hotspot occurs within a region that was previously empty. Multiple space server processes can be co-located on the same host to avoid underutilization.

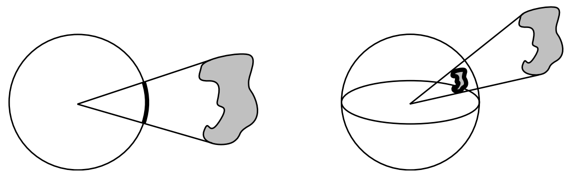

(a) Planar angle (b) Solid angle

Figure 4: Solid angle is the extension of a planar angle to three dimensions. It is defined as the area of an object projected onto a unit sphere centered at the observer.

5. GEOMETRIC OBJECT DISCOVERY

Once connected to the space, an object queries the geometric object discovery service to learn about other objects (Figure 2(c)). Due to memory, network, and display constraints of clients, the server can only return a limited subset of the world. Systems today commonly use distance queries, displaying objects within some distance d, which not only limits what a user can see and interact with, but also leads to jarring discontinuities.

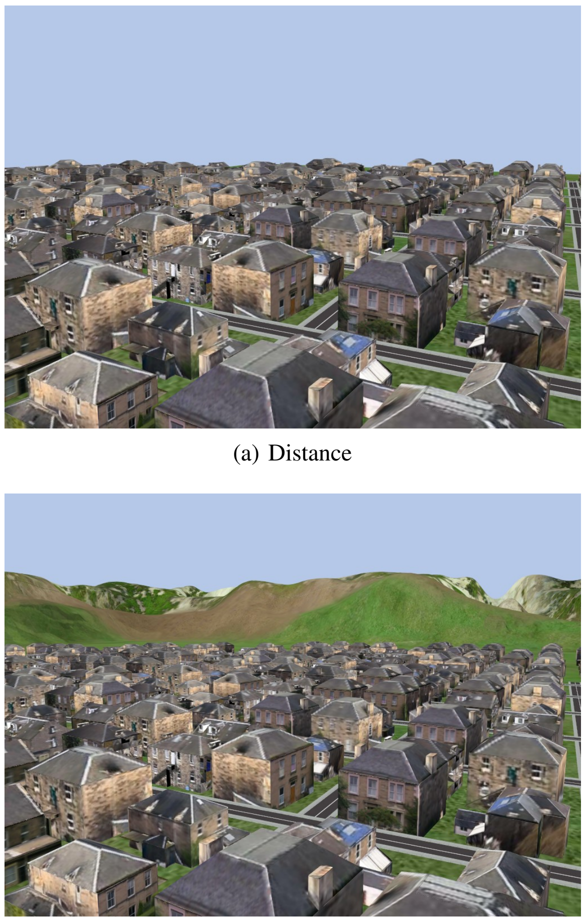
Sirikata relies on two properties of the real world to provide a better set of results than current systems:

Visual perspective: Sirikata uses solid angle queries to prioritize objects that are visually more important. The solid angle of an object is the surface area covered by the object when it is projected onto the unit sphere centered at the observer (Figure 4). It is a measure of how large an object appears to an observer and is roughly equivalent to the fraction of pixels the object would occupy if the display rendered the world in all directions at once. Figure 5 shows the substantial difference between distance and solid angle queries.

Object aggregation: Large collections of small objects, e.g., distant trees, cannot be seen individually, but together contribute significantly to a scene. Sirikata generates simplified aggregate meshes to represent aggregate objects that would otherwise be omitted and the query processor returns these aggregates along with individual objects.

Objects register a query with a simple message specifying the minimum solid angle, i.e., smallest apparent size, of objects that the server includes in the results. The query implicitly includes the querying object's current position. Because solid angle is affected by the querier position and object position and size, query results change due to querier movement, other objects moving or changing size, or objects entering or leaving the world.

Two challenges arise in implementing the solid angle query processor. The first is adapting existing data structures so solid angle queries can be efficiently evaluated


(a) Distance

(b) Solid Angle

Figure 5: Solid angle queries returning the same number of objects as a distance query give a more complete view of the world.

on a single host. Section 5.1 presents a novel data structure, the LBVH, that extends Bounding Volume Hierarchies (BVH) by having internal nodes track the largest objects in their subtrees. The second challenge is that solid angle queries are global: they can return an object from anywhere in the world if that object is large enough. Section 5.2 presents a distributed query processor, which addresses the computational and network costs of global queries spanning multiple space servers.

5.1 Efficient Solid Angle Queries

Sirikata efficiently evaluates solid angle queries by extending bounding volume hierarchies, a data structure commonly used in graphics systems for other geometric queries such as distance queries. This section describes this extension, how the same data structure handles standing queries, and how aggregate objects are returned in result queries when individual objects are too small to be returned.

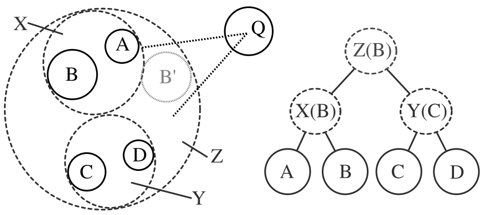


Figure 6: The LBVH tracks the largest leaf of each internal node, allowing it to test query Q against a much smaller virtual object B' instead of Z.

5.1.1 *Solid Angle Queries*

Bounding volume hierarchies recursively enclose collections of objects in bounding volumes. This enables efficient query evaluation because entire subtrees are culled if their bounding volume does not satisfy the query [2]. They form a better basis for solid angle query processing than spatial subdivisions (such as binary space partitioning [31] and kd-trees [4]) for three reasons. First, they test each object at most once because each leaf stores an object instead a list of objects overlapping the leaf's region. Second, insertions and removals are efficient regardless of object size. Finally, BVHs support an inexpensive "refit" [23] operation for when objects move: the topology of the tree remains fixed, but the bounding volumes of internal nodes are updated to account for the moving objects.

A bounding sphere hierarchy is a natural starting point because computing the solid angle for spheres is fast and easy. However, bounding sphere hierarchies are inefficient for solid angle queries because they are so conservative: a parent bounding sphere can be much, much larger than its children if those children are distant from one another. In Figure 6, for example, the bounding sphere X is much larger than either A or B. In a simple test using objects collected from Second Life regions, for example, we found evaluating solid angle queries with a BVH visited up to 86% of the tree even when fewer than 20% of leaves were in the result.

Sirikata improves the efficiency of solid angle queries through a new data structure, called the *Largest Bounding Volume Hierarchy* (LBVH). Each LBVH node includes a reference to the largest leaf in the corresponding subtree, as shown in Figure 6. When evaluating whether to traverse a subtree, the query processor tests whether the largest object in that subtree, placed as close to the querier as possible within the bounding volume, would satisfy the query. This test represents the worst case object in the subtree, however, Figure 6 shows that this test is much less conservative (e.g., B' is much smaller than Z). Using an LBVH reduces the cost of a query by 75–90% compared to a BVH.

An LBVH degrades in quality over time as objects grouped in a subtree move apart. To prevent this degradation (and need for possible full LBVH recomputation) the query processor exploits the fact that, just as in the real world, most objects are stationary. A metaverse filled with constantly moving objects is overwhelming, distracting, and difficult to navigate. Data collected from Second Life (Figure 7) show that over 95% of objects in a 4km^2 region do not move over a one hour period. Therefore, the server maintains two LBVHs, one for static and one for dynamic objects. Objects that do not move for one minute are static. The static LBVH rarely changes and so remains efficient. Because so few objects are dynamic, constant factors dominate, and a suboptimal LBVH does not significantly harm performance. The server rebuilds both trees infrequently, currently once an hour.

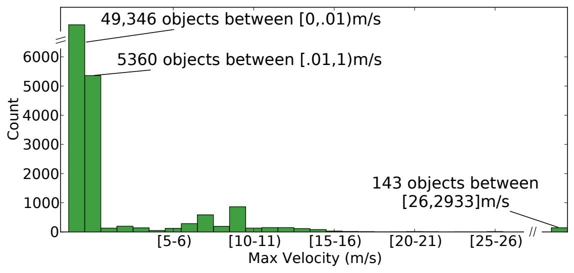


Figure 7: Most objects in a 4 km^2 region of Second Life were not moving. Outlying object speeds are due to objects teleporting.

5.1.2 *Standing Queries*

The LBVH reduces the cost of evaluating queries one time, but most queries are *standing*: after an initial result, the discovery service sends updates when the result set changes. Simple periodic re-evaluation is wasteful as the same nodes must be re-evaluated even if they have not changed. The query processor avoids this waste by maintaining cuts in the LBVH tree. Each cut stores the LBVH nodes where a query's evaluation terminated. For instance, a cut may store nodes A, B, and Y in Figure 6 if A and B satisfy the query but Y did not. Instead of restarting the query at the root node, the query processor updates the query by traversing its cut nodes. On each update, it tests whether the children of a cut node now appear large enough to satisfy the query or whether a cut node itself now appears too small to satisfy the query, updating the cut and sending notifications to the querier in both cases.

5.1.3 *Object Aggregation*

Although solid angle queries select better objects, they miss large collections of small objects. For example, Figure 5(b) is actually missing a forest full of trees which the

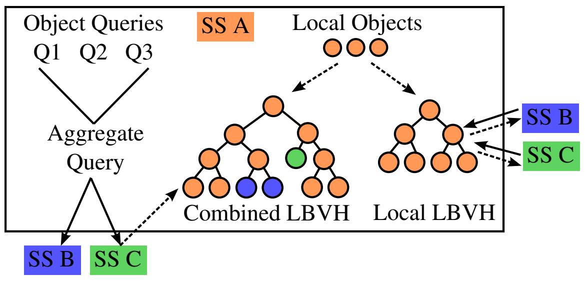


Figure 8: In the distributed query processor, servers query local LBVHs on other servers and incorporate the results into a combined LBVH. The combined LBVH resolves individual object queries.

solid angle query does not capture. To address this, Sirikata performs *aggregation*. The LBVH inherently clusters objects: the server treats each internal LBVH node as an aggregate. Each aggregate is assigned an object identifier and its meshes are combined, simplified, and stored on the CDN. By returning an entire cut instead of only leaves, the querier always has a complete view of the world, although possibly at reduced quality. Aggregates are only visual placeholders; applications cannot communicate with them.

5.2 Distributed Solid Angle Queries

The LBVH evaluates queries locally, but solid angle queries are global. Objects on any server could satisfy a query. Sirikata's distributed query processor addresses the computational and network costs of these distributed global queries. It aggregates queries to reduce inter-server communication and uses a geometric server discovery service to reduce inter-server querying. Figure 8 shows the query processor components of one server.

5.2.1 Aggregate Queries

Sirikata exploits the spatial locality of its queriers by conservatively aggregating its local queries into a single outgoing query. The aggregate query's solid angle is the minimum of every query issued to the space server. Instead of a single position, the querier is represented by a bounding sphere of the individual querier positions. To evaluate the aggregate query against an LBVH node, a virtual querier is placed within the bounding sphere as close as possible to the object being tested. Despite being conservative compared to the original queries, having only a single outgoing query reduces computational load on other servers as well as communication cost because it returns only a single set of results.

Each server maintains two LBVHs: local and combined. The local LBVH contains only local objects in the server's region. The combined LBVH includes both local objects and the results of outgoing queries, representing the set of objects in the entire world that might

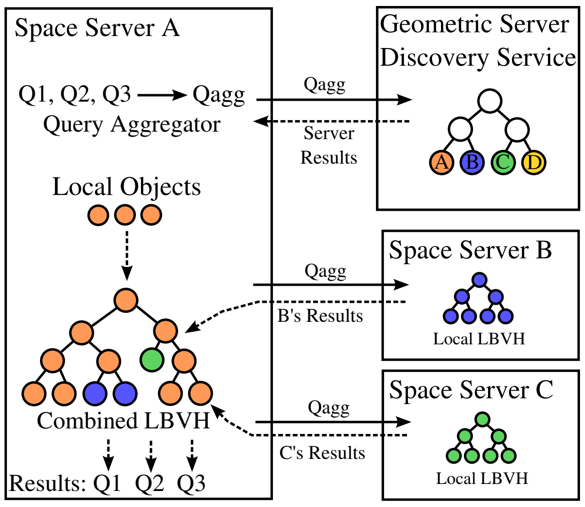


Figure 9: The geometric server discovery service determines which servers must be queried and local LBVHs resolve aggregate queries. The combined LBVH resolves individual object queries.

satisfy queries from objects within the server's region. The server resolves aggregate queries from other servers with the local LBVH and queries from local objects with the combined LBVH.

5.2.2 Geometric Server Discovery

Even with aggregate queries, N servers require N^2 server pairs to communicate because queries are global. However, due to distance and object demographics, most servers will have no results for each other. To avoid the waste of N^2 connections, servers discover which other servers to contact through a geometric server discovery service. Figure 9 shows how a server interacts with this service as part of issuing outgoing queries. The geometric server discovery service, like the object discovery service, uses an LBVH. In this LBVH, each leaf points to a space server that can further resolve the query, rather than a single object. Space servers register their aggregate query with the server discovery service and receive a stream of updates specifying which servers to contact whose objects can satisfy their queries.

To evaluate the benefits of server discovery, we simulate a world tiled with objects from Second Life traces. The world grows up to 100km^2 while maintaining constant object density. It is partitioned as described in Section 4 with up to 5000 objects per server and 100 randomly placed queriers per server. Each server's aggregate query is evaluated against every other server's objects to determine if the server must be contacted. Figure 10 shows the results for several query angles, specified in pixels occupied on a 1024x1024 screen with 60° field of view. The number of servers contacted grows more slowly than the total number of servers, eventually

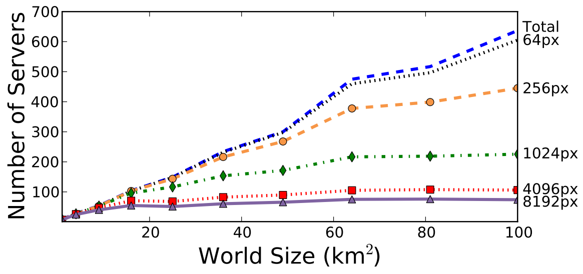


Figure 10: Average number of servers contacted with aggregate queries. Common query angles (1024 to 4096 pixels) require far fewer than the maximum.

reaching a maximum value since the world is growing in size, but not density. Currently, the server discovery service runs on a single host because a single host can easily support thousands of space servers and regenerating its state in the case of failure takes at most a few seconds.

5.3 Evaluation

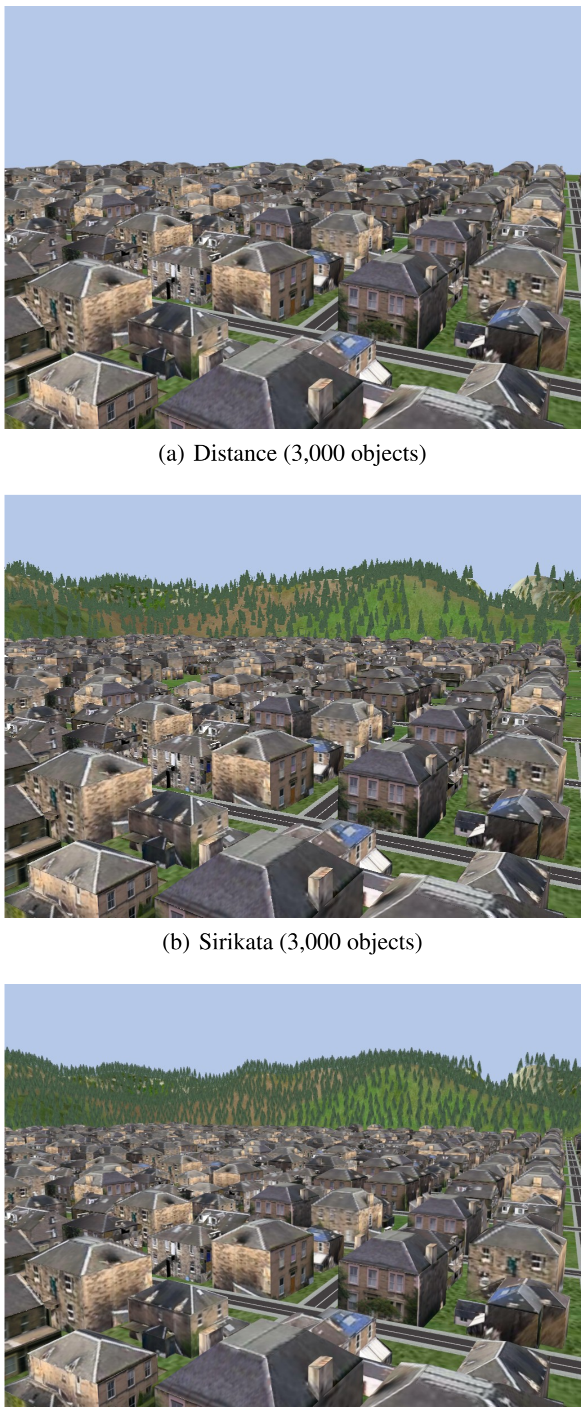
Figure 11 evaluates the visual effect of using solid angle queries, with a small town scene with 10,000 objects — houses, streets, terrain, and trees. The figure shows images with ~3,000 objects for distance queries and Sirikata's queries as well as the full image (all 10,000). Distance queries miss important objects like the terrain. As Figure 11(b) shows, solid angles with aggregates complete the picture, including detail — such as the distant forest — that is lost otherwise, allowing a client to see a compelling approximation of the entire world.

6. ROUTING AND FORWARDING

Returning to Figure 2(d), after an avatar discovers the building, it interacts with it through application messages. Space servers are responsible for delivering these messages to the proper object host. This delivery has two steps: routing, to determine which space server covers the region the destination is in, and forwarding, delivering the message to that server. Section 6.1 describes routing; Section 6.2 describes forwarding.

6.1 Application Message Routing

Two properties shape the application message routing table's design. First, any pair of objects should be able to communicate, even distant ones. Unlike games, which can be carefully engineered to use only local messaging, metaverse applications may need to communicate over long distances. Second, objects may move freely about the world. Long-range object communication requires that the routing table on any space server be able to answer a query for any object in the world. Object mobility suggests the use of a flat object identifier namespace.



(a) Distance (3,000 objects)

(b) Sirikata (3,000 objects)

(c) Ideal (10,000 objects)

Figure 11: Comparison of Sirikata (solid angles w/aggregates), distance-based queries, and the ideal scene. Sirikata's combination of solid angle queries and aggregation allows it to display the complete world with a fraction of the objects.

A simple key-value lookup, mapping from object identifier to destination server, satisfies these requirements. Sirikata stores routing records in a separate scalable, key-value store [29]. This approach benefits from simplicity, reusing existing software to maintain a single, globally consistent routing record for each object.

To mitigate the cost of a round trip to the backend store for each message, Sirikata heavily caches routing records on each space server, relying on three factors. First, each record is small, only 34 bytes, containing the object's identifier, the space server it is connected to, and metadata the forwarder uses to weight flows. Millions of records can be cached in only tens of megabytes. Second, object mobilitiy is limited. Like in the real world, few objects move in the virtual world (Figure 7). Correspondingly, their routing records do not change. Finally, by caching records when objects migrate, a server can cheaply forward messages it receives due to stale routing entries. It also sends a control message back to the source space server with the new routing record.

6.2 Application Message Forwarding

Once a server has the next hop for a message in its routing table, the forwarder is responsible for deciding when to send it. Under low load, forwarder behavior is simple — forward all packets — but as demand exceeds capacity, it must choose the order in which to forward packets and which packets to drop. Unlike games, where there is only one application that can be designed around network limitations, Sirikata must support many concurrent applications with unknown traffic patterns.

Sirikata draws inspiration from the real world by weighting inter-object flows using an equation similar to the falloff of electromagnetic radiation. This gives closer and larger object pairs a greater portion of bandwidth even under heavy congestion. The challenge of this approach is that it requires distributed state spread across many servers (objects' positions and sizes) and traditional techniques for enforcing these weights are too expensive as the number of communicating object pairs grows.

As discussed and evaluated by Reiter-Horn [18], Sirikata uses heuristics to approximate object positions and sizes and leverages ideas from Core Stateless Fair Queueing to enforce weights it assigns between flows. Reiter-Horn demonstrates that four properties of the Sirikata forwarder provide a much more natural, intuitive experience for users. First, weights are always non-zero, so objects are always able to communicate. Second, just as two people step closer to hear more clearly, objects can simply move closer for higher throughput because weights fall off gradually with distance. Correspondingly, throughput drops as objects separate, but never drops suddenly. Third, through careful selection of the weight function, a minimum quality of service is guaranteed between a pair of fixed objects, even if all other object pairs are trying to communicate at full capacity. A close pair of objects cannot be drowned out. Finally, the forwarder ensures high utilization: if there is only one flow, it can use the full server capacity.

	Latency
Space Server to Upper Tree Latency	1498 μs
Upper Tree Lookup	24 μs
Upper Tree to Lower Tree Latency	848 μs
Lower Tree Lookup	137 μs

Table 2: Latency breakdown for partitioning service queries with a cut depth of 20 on a LAN.

7. PERFORMANCE EVALUATION

Previous sections motivated and evaluated the design decisions in each of the server's services. This section evaluates the performance of their implementations, as well as the end-to-end performance of the entire server. Evaluations were performed on a cluster with 2.4GHz Xeon E5620 CPUs, 8GB RAM, and 1Gbps NICs.

7.1 Implementation

The core Sirikata system is currently ~102,000 lines of code. The space server and services described in this paper are ~22,000 lines of code. The object host, including the graphical client and scripting libraries, are ~45,000 lines of code. The remaining ~35,000 lines of code are in shared utilities. To leverage multicore processors, the space server is highly multithreaded: the current implementation has eight active threads.

7.2 Partitioning Service Performance

The key metric for the partitioning service is latency: how quickly can the system determine the servers responsible for a point or region? Table 2 shows the breakdown of latency for a query using the world population data described in Section 4 and a cap of 40 objects per server. The table separates the two network hops because the server used to query the upper tree is unlikely to contain the lower tree covering the queried region. The latency is ~2.5 ms, of which ~2.3 ms is two RTTs of network latency.

7.3 Geometric Query Performance

Figure 12 shows the query update rate of Sirikata's geometric object query processor, in queries per second, for a world generated from measured Second Life data. The scene contains 100,000 static objects and 1,000 queriers, generated by tiling Second Life data at a higher density to cover $\frac{1}{4}$km^2. All queriers use the same solid angle parameter and follow measured avatar paths.

Cuts reuse query state from the previous iteration and

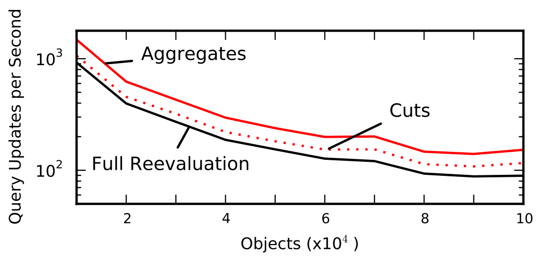


Figure 12: Query updates per second, as the number of objects increases. Cuts improve query update rate by 20% over full reevaluation and cuts with aggregates improve the rate by 56%.

	Latency	Max Rate	Throughput
Local	324 μs	144,632 pps	384.18 Mbps
Remote	672 μs	71,074 pps	320.25 Mbps
Remote+Lookup	974 μs	38,775 pps	165.79 Mbps

Table 3: Space server forwarding performance. The system can enforce fairness at fine granularity without sacrificing performance.

substantially improve performance over periodically reevaluating full queries, updating about 20% more queries per second. Aggregates improve performance further, with 56% higher query throughput than full reevaluation. Aggregates avoid many solid angle tests at leaves: once one leaf is included in the cut, all its siblings must also be included.

7.4 Routing and Forwarding Performance

We evaluate router and forwarder latency, forwarding rate, and throughput with microbenchmarks between pairs of objects. Latency measures the application-level ping time for 64-byte messages with idle servers. Forwarding rate measures how fast a server can forward 64-byte messages. Throughput measures the maximum inter-object throughput using 1KB messages.

We measure three forwarding paths: local, remote, and remote+lookup. The local path handles messages between objects connected to the same server. The remote path handles messages between objects connected to different servers where the destination routing entry has been cached, passing through inter-server queues and requiring an additional network hop. The remote+lookup path is like remote but also requires a routing table lookup.

Table 3 shows the results. For local messages, the space server can process about 145,000 messages per second and has a latency of about 325μs. A routing service lookup triples message latency, cuts the forwarding rate by 75%, and reduces throughput by 57%. This demonstrates the need for a routing table cache. These demonstrate that the system can enforce fairness while maintaining high performance.

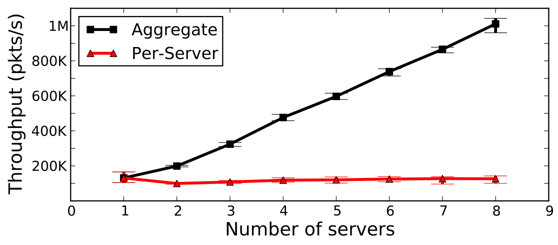


Figure 13: Egress capacity of the space as the number of servers increases, using a uniform workload. Values are the median of a 5-minute period, with 99th percentile error bars.

7.5 Scaling Performance

We examine how server performance scales when exercising all of the services under a simple, general workload. This workload consists of objects with a random size, position, and solid-angle query parameter. 95% of the objects are stationary and the rest periodically teleport to a random position and set a random velocity. All objects send 150 30-byte messages per second to a random object they can see. Each server covers a 10km by 2km by 4km regions and servers are added linearly. Object density remains uniform: more objects are added with each additional server.

Figure 13 shows the results for up to space servers with many more objects hosts generating traffic. As the size of the world increases, the aggregate egress capacity of the world increases roughly linearly. While objects in this workload can and do interact with distant objects, e.g., interesting buildings, most of their interactions are with nearby objects, as one would expect. The small drop from one to two servers is due to routing some traffic across server boundaries. While this workload may not exactly model a real-world scenario, it shows that the server can scale up to larger spaces without limiting either a client's view of the world or its ability to communicate with distant objects.

8. APPLICATION EVALUATION

Performance and scalability are only useful if the ability to see and interact with the entire world leads to engaging applications that are not possible in today's systems. We hired 11 undergraduate computer science students for the Summer of 2011 to develop applications in Sirikata. In a few weeks, they created a variety of applications and systems, including:

Minimap - a top-down display of objects and events,

Wiki-world - embedded Wikipedia articles from object metadata,

Escrow - secure object and virtual currency exchange,

Games, including a Pokemon-based RPG, a 3D Pacman clone, 3D tower defense, and a Minecraft clone,

Procedural generators for both city road networks and building layouts, and

Phys-lib, a library for customizable physics with an example billiards application.

These applications demonstrate that Sirikata supports a variety of metaverse applications running concurrently, including some that would be very difficult or impossible in current systems. We focus on two examples, Minimap and Wiki-world.

8.1 Minimap

Minimap presents a 2D top-down view of objects and events as thumbnails. Geometric queries seed the map and it uses messaging with other objects to register events and descriptions. Events may include code to be executed by the avatar, for example to register it as a player in a game. Users can also specify regions to aggregate when zoomed out and share them with others, for example assigning a neighborhood a name. Solid angle queries and long-distance messaging make Minimap feasible, allowing it to give a view of the world beyond its immediate surroundings. In contrast, Second Life requires a separate, centralized 2D map service, which users cannot extend or improve.

8.2 Wiki-world

Wiki-world is similar to the application in Section 1: users can click on objects to learn about them through embedded Wikipedia articles. Executing within an avatar's script, Wiki-world collects search terms by messaging the object and from user-specified tags on the CDN, and presents search results in a 2D interface. The Sirikata server enables Wiki-world for the entire world by allowing discovery of, and communication with, large, distant objects such as the building in Figure 2.

9. RELATED WORK

Sirikata builds on a combination of ideas from distributed systems, networking, and graphics. This section compares Sirikata to existing metaverse systems and reviews prior work that informs the design of its services.

9.1 Metaverses

Metaverses differ from most application-specific virtual worlds, such as those for games and visualization, because they present a single, contiguous world comprised of user-generated content. Scalability tricks used in those systems do not translate directly to metaverses. Most metaverses, including ActiveWorlds [1], Second Life [30] (as well as its open source counterpart OpenSim [27]), and Blue Mars [7], have made the same compromises to enable scalability, choosing to limit visibility and interaction to a small distance in order to scale. The distributed scene graph [22] improves the scalability of OpenSim by running system services across multiple hosts. However, the limits on visibility and interaction are still present.

9.2 Partitioning Service

A few early virtual world games, such as Asheron's Call, dynamically partitioned the world across a cluster of servers [28]. However, most systems use sharding or precomputation to achieve scalability. World of Warcraft divides users into hundreds of "realms," or copies of the world [39]. RING models the world as a large piece of global state, but prunes object updates based on precomputed visibility within a static environment [15]. These techniques do not apply to metaverses where there is a single, contiguous world with dynamic, user-generated content.

Liu et al. make design choices similar to Sirikata in applying binary space partitioning for managing load [24], but no global data structure is maintained. Chen et al. use fixed partitioning, but exploit and encourage local communication by clustering neighboring regions onto hosts [9].

9.3 Discovery Service

Practically all systems today limit object discovery by distance, sometimes referred to as range queries or area of interest (AOI) [6, 20]. Other systems use application-level information [5], disjoint region splitting [21], perceptual limitations [32], or visibility [36] to reduce load. Chaudhuri et al. describe a system for rendering large worlds, guaranteeing a fixed bandwidth cost, given a maximum velocity [8]. All these approaches assume either local interaction or a static environment. Sirikata's discovery service builds on a long history of spatial data structures. It extends the the well-understood bounding volume hierarchy [11] and applies it in a distributed setting to perform efficient, global, solid angle queries.

9.4 Routing and Forwarding

Routing and forwarding is a minor concern in most current metaverse systems because local discovery limits communication to local objects. Sirikata enables communication between any pair of objects. Although most services in Sirikata leverage geometry, routing uses a flat namespace of object identifiers to break any dependency on the assumption of stationary objects. Existing techniques [38], such as distributed hash tables [34, 13], are used to scale identifier lookups. Since object changes often have many observers, many systems use multicast to reduce network traffic [14, 17, 15]. Sirikata focuses on unicast application-level traffic, although multicast could be applied for sending location table updates. Sirikata's forwarder builds on core stateless fair queueing (CSFQ) [35] to manage weighted flow fairness with minimal state [18].

10. CONCLUSION

This paper argues that metaverses have failed to meet their imagined potential: rather than large, vibrant social spaces, they are quiet, lifeless, and desolate. In order to scale to support a large world, existing systems prevent users from seeing that world, constraining them to a small, local area.

The Sirikata metaverse server allows users to see and interact with a large and dynamic virtual world. Properties of the real world inform its design and provide constraints that enable scalability with intuitive implications: observers have limited resolution, objects are mostly static, and object density is in a small, fixed range. Sirikata enables applications that are very difficult to build in existing systems without sacrificing performance.

The Sirikata metaverse server is one component of a larger vision. Many challenges arise in the object host and CDN described in Section 3, from load balancing objects in a distributed object host to supporting progressive loading of content from the CDN. We believe distributed graphical systems are an underexplored area of research and significant work remains at the intersection of the graphics and systems communities. As computing increasingly moves into the physical world, in the form of augmented reality, cyber-physical systems, and ubiquitous computing, we believe the principles Sirikata has begun to explore may inform the design and implementation of these increasingly important classes of systems.

Acknowledgments

Thanks to our undergraduate developers and testers — Kotaro Hoshino, Jiwon Kim, Elliot Conte, Angela Yeung, Kevin Chung, Emily Ye, Alex Quach, Ben Christel, Will Monroe, Zhihong Xu, and Gabrielle Chen — for spending their summer building applications in Sirikata. Thanks also to Xiaozhou Li, Bhupesh Chandra, Lily Guo, and Patrick Horn for their valuable contributions to Sirikata. We would also like to thank the anonymous reviewers and our shepherd, John Regehr, for their helpful comments.

This work is funded by the National Science Foundation (NeTS-ANET Grants #0831163 and #0831374) and conducted in conjunction with the Intel Science and Technology Center – Visual Computing.

11. REFERENCES

[1] `http://www.activeworlds.com/`.

[2] J. Arvo and D. Kirk. A survey of ray tracing acceleration techniques. *An Introduction to Ray Tracing*, 1989.

[3] H. Bennetsen. Sirikata: Old stuff in new ways. Lecture at Media X at Stanford University, November 2009.

[4] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9), 1975.

[5] 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 *SIGCOMM Comput. Commun. Rev.*, Aug. 2008.

[6] A. Bharambe, J. Pang, and S. Seshan. Colyseus: a distributed architecture for online multiplayer games. In *Networked Systems Design and Implementation*, May 2006.

[7] `http://www.bluemars.com/`.

[8] S. Chaudhuri, D. Horn, P. Hanrahan, , and V. Koltun. Image-based exploration of massive online environments. Technical Report CSTR 2009-02, Stanford, 2009.

[9] J. Chen, B. Wu, M. Delap, B. Knutsson, H. Lu, and C. Amza. Locality aware dynamic load management for massively multiplayer games. In *Principles and Practice of Parallel Programming*. ACM, 2005.

[10] Gridded population of the world, version 3. `http://sedac.ciesin.columbia.edu/gpw`. Socioeconomic Data and Applications Center (SEDAC), Columbia University.

[11] J. H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 1976.

[12] E. Coumans. Bullet physics engine. `http://www.bulletphysics.org`.

[13] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a DHT for low latency and high throughput. In *Networked Systems Design and Implementation*, 2004.

[14] E. Frécon and M. Stenius. Dive: a scaleable network architecture for distributed virtual environments. *Distributed Systems Engineering*, 1998.

[15] T. A. Funkhouser. RING: A client-server system for multi-user virtual environments. In *Symp. on Interactive 3D Graphics*, Apr. 1995.

[16] T. A. Funkhouser and C. H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. ACM SIGGRAPH, 1993.

[17] L. Gautier and C. Diot. Design and evaluation of MiMaze, a multi-player game on the internet. *ICMCS*, June-July 1998.

[18] D. R. Horn. *Using a Physical Metaphor to Scale Up Communication In Virtual Worlds*. PhD thesis, Stanford University, 2011.

[19] M. Hoybe. BE community: Bridging social isolation for teenagers & young adults with cancer. Games for Health, 2011.

[20] S.-Y. Hu, J.-F. Chen, and T.-H. Chen. Von: A scalable peer-to-peer network for virtual environments. *Network, IEEE*, 2006.

[21] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *INFOCOM*, Mar. 2004.

[22] D. Lake, M. Bowman, and H. Liu. Distributed scene graph to enable thousands of interacting users in a virtual environment. NetGames, 2010.

[23] C. Lauterbach, S. Yoon, and D. Manocha. RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs. In *IEEE Symp. Interactive Ray Tracing*, Sept. 2006.

[24] H. Liu and M. Bowman. Scale virtual worlds through dynamic load balancing. In *Distributed Simulation and Real Time Applications*, 2010.

[25] B. F. T. Mistree, B. Chandra, E. Cheslack-Postava, P. Levis, and D. Gay. Emerson: Accessible scripting for applications in an extensible virtual world. In *OOPSLA*, 2011.

[26] `http://nwn.blogs.com/nwn/2010/01/empty-beauty-of-second-life.html`.

[27] `http://www.opensimulator.org/`.

[28] J. Quimby. Massively multiplayer gameplay system

implementation. Game Developers Conference, 2002.

[29] `http://redis.io/`.

[30] `http://www.secondlife.com/`.

[31] R. Shumacker, R. Brand, M. Gilliand, and W. Sharp. Study for applying computer-generated images to visual simulation. Technical Report AFHRL-TR-69-14, U.S. Air Force Human Resources Lab, 1969.

[32] S. K. Singhal. *Effective remote modeling in large-scale distributed simulation and visualization environments*. PhD thesis, Stanford University, 1997.

[33] `http://www.sirikata.com/`.

[34] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. 2001.

[35] I. Stoica, S. Shenker, and H. Zhang. Core-stateless fair queueing: achieving approximately fair bandwidth allocations in high speed networks. 28(4), 1998.

[36] S. J. Teller and C. H. Séquin. Visibility preprocessing for interactive walkthroughs. In *ACM SIGGRAPH*, 1991.

[37] The Second Life Economy in Q2 2011. `http://community.secondlife.com/t5/Featured-News/The-Second-Life-Economy-in-Q2-2011/ba-p/1035321`.

[38] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes no longer considered harmful. In *Operating Systems Design and Implementation*, 2004.

[39] `http://www.wowwiki.com/Realm`, 2011.