To Infinity and Not Beyond: Scaling Communication in Virtual Worlds with Meru

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

[†]*Stanford University* [⋆]*Princeton University*

Abstract

Virtual worlds seek to provide an online setting where users can interact in a shared environment. Popular virtual worlds such as Second Life and World of Warcraft, however, rely on share-nothing data and strict partitioning as much as possible. They translate a large world into many tiny worlds. This partitioning conflicts with the intended goal of a virtual world by greatly limiting interaction and reducing the shared experience.

We present Meru, an architecture for scalable, federated virtual worlds. Meru's key insight is that, compared to traditional distributed object systems, virtual world objects have the additional property of being embedded in a three-dimensional geometry. By leveraging this geometric information in messaging and caching, Meru can allow uncongested virtual world objects to pass messages with 800 times the throughput as Second Life while also gracefully scaling to handle the congestion of ten thousand active senders. Unlike virtual worlds today, Meru achieves this performance without any partitioning, maintaining a single, seamless world.

1 Introduction

Three-dimensional virtual worlds are a common feature of futurist visions and science fiction. Today's systems, however, fall far short of this imagined potential. Rather than support user applications and a seamless shared experience in enormous, rich worlds, systems like Second Life, EvE Online, and World of Warcraft enforce harsh and disconcerting restrictions, imposing a "fog" past which an object cannot see or interact with the world. Skyscrapers are invisible until you stand beneath them; two spaceships cannot communicate until within a predefined range; a user cannot control something they own unless immediately next to it. Furthermore, unlike something like the web, users cannot extend system resources: all code in the world runs on centrally controlled servers.

Computational scalability in three-dimensional environments, such as physics simulation [4] and graphics rendering [20], is a well understood problem with existing solutions. The limitations described above relate to communication: the range and rate that virtual world objects such as avatars, skyscrapers, vendors, and pets can exchange data. Imagine an "Internet" where you can only talk on your LAN and you cannot add a host. This is the state of virtual worlds today.

Virtual worlds today impose these limitations because their underlying system architectures are based on partitioned, shared-nothing designs. Yet like objects in the real world – and unlike the original shared-nothing distributed databases [30] – virtual worlds objects reside in a shared, continuous space. A virtual world resembles a large-scale distributed system of message-passing objects. What differentiates virtual worlds from prior systems, however, is that these objects and their access locality are embedded in a three-dimensional geometry.

The key insight of this paper is that a virtual world system can leverage this geometric information to provide a high performance and scalable messaging system with a much less restrictive communication model than the worlds of today. Unlike with the fixed-limit approaches taken by most modern virtual worlds such as Second Life and World of Warcraft, any pair of objects can potentially communicate at speeds greater than 10 Megabits. When the world is heavily congested with ten thousand senders or more, nearby objects continue to have reasonable throughput on the order of 30 kilobits.

We present evidence that a virtual world can achieve these properties by enforcing analogies to real-world physics. We present a message passing system where message rates are analogous to light propagation, proportional to an object's geometric size and decaying with distance. The system guarantees a messaging rate between two objects based on a geometrically derived falloff function. Although guaranteed rates between nearby and larger objects are greater than between

smaller and more distant ones, there are no hard restrictions on object communication. If there is excess capacity, distant pairs of objects can communicate at a high rate. This enables the virtual world to scale because, as the geometric size of the world grows to infinity, our approach ensures that communication input and output of a server converges to a constant.

This paper describes the design and implementation of the message communication layer of Meru, a federated virtual world system. It makes three research contributions. The first is the Meru virtual world architecture. Meru's federated organization permits users to run their own objects and communicate in a shared world run by a third party. This feature is unlike existing designs, where one administrative domain controls the world and runs all objects. Federation enables extensibility: it opens a Meru world to new services, as users can introduce new objects that run using computational resources they own. Meru breaks a virtual world into three parts: space servers, object hosts, and a resource content distribution network (CDN). Space servers administer and control the shared geometric space and route object messages: they are the focus of this paper.

The second contribution is a Meru space server's message forwarder. We show that by applying a geometric rate control algorithm based in real-world physics, Meru is able to dynamically allocate network capacity across a wider range of loads. While worlds like Second Life statically allocate throughput and give a pair of isolated objects at most 57kbps, Meru dynamic allocation can dedicate the full server capacity of 47Mbps, an 800-fold increase. Meanwhile, as contention increases to thousands of competing object pairs, Meru is able to gracefully scale throughput so no pair is starved and nearby objects receive 30kbps. Meru also cuts object message latency by 96% compared to Second Life (1.2ms vs. 33ms).

The third contribution is OSeg, a chain-replicated key-value store that maps objects to space servers. OSeg's research contribution lies in its geometrically-based least forwarder weight (LFW) caching algorithm. The geometric nature of virtual worlds means that simple popularity metrics do not always apply, and mobility is a major concern. Compared to a standard LRU cache, LFW is up to 75% closer to an optimal oracle cache.

Because virtual worlds are a nascent technology, there are no well-known or well-accepted representative workloads. We therefore evaluate Meru using four distinct workloads, which we believe capture major possible uses: a social model, where object pairs communicate with the small-world distributions typical to social networks; a content model, where object pairs communicate using a Zipfian distribution; a graphical model, where object pairs communicate based on ray-tracing visibility; and a throughput model, where objects pairs communicate based on Meru's communication falloff function.

2 Virtual Worlds Today

In the context of this paper, virtual worlds are interactive, continuous, and shared 3D spaces. Participants appear as avatars in the space, which also contains simulated objects – anything from mountains to clocks to clothing. All objects have a physical presence and properties, such as position, geometric shape, and appearance. The world can enforce physical laws such as gravity and collisions.

In addition, objects require a mechanism for sending messages to other objects in their world. These messages allow objects to dynamically respond to events and conditions in real-time. Without some form of communication messaging, objects such as avatars and bots would not be able to interact with other users nor any features in their environment: they would not be able to eat a virtual apple, swing a virtual sword, nor plant a virtual tree. In short, virtual worlds would be little more than a boring, static, 3D environment. For this reason, the correct design and implementation of a virtual world's object-messaging pipeline is fundamental to the creation of truly engaging and immersive spaces for users.

At a high level, a virtual world resembles a system of distributed, message-passing objects. However, unlike typical distributed object systems, which use naming or lookup services, virtual world objects discover each other through geometric proximity. Where objects in traditional systems are organized in hierarchies like file systems or CORBA compound names, organizing objects in a virtual world is like organizing them in the real world: they are placed near each other and arranged in a way that is easy to browse.

2.1 Three Example Worlds

Numerous virtual worlds today fit our description above. We present three canonical examples, describe what restrictions they impose, and argue how these restrictions prevent these worlds from being effective application platforms. These limitations commonly arise from their architectures based on shared-nothing, partitioned state.

Second Life [22] is a general virtual world platform. Users can create and script new objects which run on Second Life servers. Second Life is a single, continuous world, divided into 256 m x 256 m meter regions. Regions are statically bound to servers: there is no migration or load-balancing [33]. Because of this limitation, most regions can hold only 40 avatars, which often requires large events to occur at the intersection of 4 regions [18]. Neighboring regions share only minimal state about objects. Because of this, clients never truly

view the world: they can only see and interact with objects within a small distance. Scripted objects discover only 16 nearby objects at a time and interact through short range "say" messages: longer range communication is via rate limited HTTP requests to handled by objects [23].

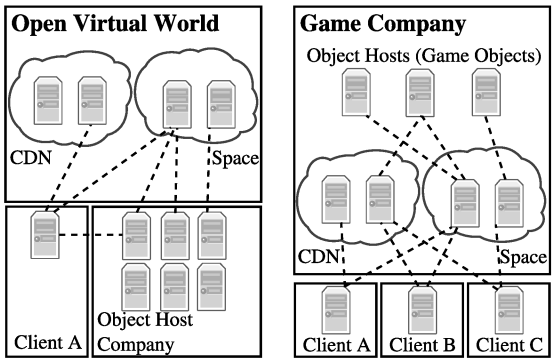
World of Warcraft (WoW) splits its 11 million users across 772 replicas [37, 3]. Each replica is a cluster of servers [38, 39]. Each of the world's 4 continents is a seamless virtual space run on a single server, but most core game content is in "instances," partitioned regions that run on separate servers. Spells in WoW reach up to 40 yards, and while terrain can be viewed up to 1,277 yards away, objects can only be seen up to a few hundred yards away, leaving the distant world looking barren. WoW controls all content: it is not possible to add new objects. When a replica is overloaded, players wait in login queues as long as 1,000 clients (a few hours).

EvE Online is a multiplayer game set in space. EvE has a single shared galaxy, partitioned into isolated solar systems that are load-balanced on a custom high performance cluster. EvE has a complex electronic warfare system that governs visibility and targeting, with limits of approximately 100km. Like WoW, users cannot add objects. Popular solar systems (such as Jita) can have significant latency, which introduces a user feedback loop to prevent overcrowding [12, 10]. Users can notify administrators of a planned conflict to pre-dedicate a server to the solar system in question [36].

WoW, EvE, and other game platforms have parallels with the online access providers of the early 1990s (Prodigy, AOL). They are vertically integrated and centrally controlled content platforms. Experience has shown that open platforms which empower users – such as the world wide web – lead to innovative applications and technological growth. Second Life supports user generated content, but scripts are highly constrained and centrally hosted. Furthermore, its population and distance limits make it a poor environment for user interaction [19]. Despite a huge surge in interest a few years ago, Second Life today is mostly empty, with only a few fringe groups responsible for most activity.

We believe that three properties will enable virtual worlds to take a critical step closer to rich, shared environments:

- **Federation**: just as anyone can extend the web by adding a host to serve pages, users should be able to add computational resources to run objects.
- **Communication**: the world should not have coarse, disconcerting distance limitations: any pair of objects should be able to communicate, with reasonably high throughput and low latency.



(a) Open Virtual World (b) Game

Figure 1: Two ways to deploy Meru. In (a), one company runs the space and CDN for an open social virtual world, while third parties provide their own objects. In (b), a game company runs all components except for clients.

- **Scalability**: worlds should be able to grow to enormous sizes with huge numbers of objects while meeting the communication and federation requirements listed above.

These are only a subset of the technical challenges in a next generation virtual world platform. However, as they address some of the most fundamental limitations in existing systems, we believe they are the right first step.

3 Federation: The Meru Architecture

The Meru architecture supports federated, application-rich virtual worlds by separating a virtual world into three individually administered parts: space servers, object hosts, and a content distribution network (CDN). This section describes the architecture, focusing on space servers, which control and govern communication. The rest of the paper deals with the two core space-server communication services, the message forwarder and the object-to-server map (OSeg). This section provides the context in which these two services execute.

3.1 Spaces, Object Hosts, and a CDN

Existing virtual worlds tightly couple system design with application-level properties. For example, EvE partitions the universe into solar systems. While suitable for closed, commercial products, this integration does not lead to general systems principles that can be applied to a wide range of virtual worlds. Although the application-level properties of a vast desert planet may be vastly different from what we would expect in a world composed of a single megalopolis, both may be built up from the same systems components, primitives, and abstractions.

The Meru architecture breaks this coupling by separating object execution from world simulation. Meru splits a virtual world into space servers, object hosts, and a content distribution network. A Meru world – a "space" – is quite literally an address space: objects have unique identifiers as well as geometric coordinates. The space is the final authority on what objects are in it, their location, and their physical properties. The space also handles geometric queries for object discovery and routes messages between objects. A given space is run by one or more **space servers**, which segment the geometric coordinates of the 3D world. All of the space servers for a world are under a single administrative domain.

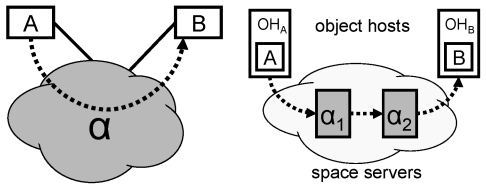
Responsive and animate objects are key aspects of a compelling virtual world: a dog should bark when someone attempts to steal from its master's virtual home; a flower should grow and blossom; a machine gun should run out of ammunition as it fires. To provide for such engaging objects, objects have associated scripts that specify their behavior. Unlike most commercial systems, Meru federates object scripting, creating a separate entity, the **object host**, specifically tasked with executing object code. Object hosts run object scripts while connected to space servers. Space servers in turn route location updates and message traffic back to the object host.

Although potentially dynamic and changing places, virtual worlds still have much large, static content. For instance, in a rich visual world, an object's mesh/texture could easily be several megabytes. Given that many avatars may access these large meshes and that latency can have a profound effect on a user's perceptions [7], the Meru system incorporates a **content distribution network**, which serves large data resources. Objects do not communicate these large data items directly: they pass references to elements in the CDN. The CDN offloads high bandwidth communication from space servers and object hosts, while providing a natural way to manage replication and accessibility of commonly used resources (*e.g.*, a particular vehicle model).

This decomposition allows the Meru architecture to support federated worlds as well as more traditional applications. Users can run objects on hosts they control, yet interact in a neutral space, as in the open world of Figure 1(a). Similarly, a game company can run both object hosts and space servers, completely controlling the code in its world, as in Figure 1(b).

3.2 Space Server Responsibilities

For two Meru objects to interact, they must be in the same space and exchange messages. Space servers mediate all inter-object communication, and objects interact with the world by directly sending messages to the space. For example, a movement command from an object goes to the space, while a request to open a door passes through the space and goes to the door. Because a space is segmented across multiple space servers, a message between objects may pass through two servers. Figure 2(a) shows an example of the logical operation of object A sending a message to object B within space α. Figure 2(b) shows how this can map to servers.



(a) Logical (b) Physical

Figure 2: Inter-object messaging. Logically, a message from A to B passes through space α; in the system, the message passes from object host OH_A to space server α_1, to space server α_2, to OH_B.

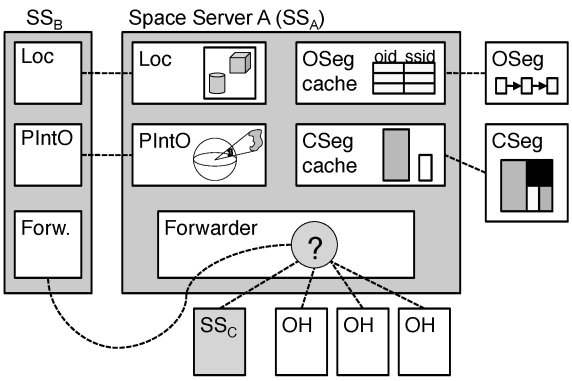


Figure 3: Space server internals. Dashed lines show network connections.

A Meru space has four basic responsibilities. First, it routes and forwards messages between objects. Second, it maintains the authoritative position and other geometric properties of objects by handling physics and movement requests. Third, it answers geometric queries about what other objects are nearby. Finally, it provides audio streams of the environment. Meru space servers handle the first three: special audio-mixing hosts handle the last.

3.3 Example Execution

Figure 3 shows the internals of a Meru space server. To demonstrate how these services coordinate to provide a virtual world with scalable communication, we present an example of an object O entering a world and communicating with another object. We assume that O has been granted entry to the space and given its initial position.

O's object host must find the space server SS_O authoritative for O's position. To accomplish this task, the object host sends a query to any space server. That space server contacts the Coordinate Segmentation (CSeg) service, issuing a lookup which returns the authoritative space server ID, SS_O, for that position.

The object host connects to SS_O and registers O. Registering O puts an entry in SS_O's location table (Loc). SS_O writes an entry to the Object Segmentation service (OSeg), which maps object identifiers to their authoritative space servers.

O is now present in the world and visible to other objects. O registers a standing (streaming) query to discover other relevant objects with the Potentially Interesting Object (PIntO) service. PIntO begins streaming object identifiers and CDN references to geometric properties such as meshes. If O represents a user client, it can use these references to start rendering the scene, loading data from the CDN as needed.

Using an object identifier returned from PIntO, O interacts with it by sending a message. The object host sends this message to SS_O's Forwarder. The Forwarder uses Loc (for local objects) and OSeg (for remote objects) to determine the destination space server. The destination space server's Forwarder forwards the message to the appropriate object host, which delivers it to the destination object.

3.4 Object Messaging

This last step – sending a message – presents a basic scalability challenge and is the focus of the rest of this paper. While a space server can shed query and location update load by reducing the geometric region it covers, message input and output are not as easy to control.

As Section 2 discussed, current virtual worlds scale communication by setting hard distance limits and splitting throughput uniformly among communicating objects: a "fog" set at a fixed distance acts as a communication barrier. While such an approach is tenable for closed, tightly controlled worlds such as World of Warcraft and EvE Online, it poses major problems for virtual worlds as an application platform. Communication in these worlds has very strange behavior: objects that are at one moment reachable, cross a distance barrier, and are suddenly unreachable. Further, very large objects just outside the fog are unreachable, while tiny objects just inside are reachable. This model makes all but the most simple applications that only require short bursts of localized communication difficult or impossible to build.

The Meru architecture takes a different approach, as the next section describes, one which meets the communication and scalability requirements.

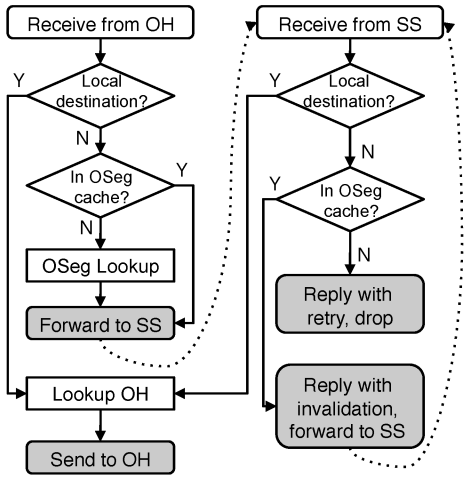


Figure 4: Flowchart of the object message forwarding pipeline in a Meru space server. The bottom of the right side is the case when a space server receives message for an object that has moved away.

4 Scalable Communication: Forwarder

This section describes the design and implementation of Meru's forwarder. When describing Meru's forwarder, we use the terms "object pair" and "flow" interchangeably.

4.1 Basics

Space servers provide a best-effort datagram service between objects. Figure 4 shows the flowchart of how a space server forwards messages between objects. A destination is local if the space server is authoritative for that object. The flowchart has three basic outcomes: send the message to the destination's object host, forward the message to another space server, or drop the message.

A space server assigns each active flow a weight. Using techniques based on weighted fair queueing [8], space servers grant each flow a share of the available capacity proportional to that flow's weight. For example, if a flow has weight 3.5 and the sum of weights is 70, that flow will receive *at least* 3.5/70 = 5% of the network capacity. If it is the only active flow, it will receive 100% of the capacity. Using fair queueing allows Meru to enforce weights while supporting intelligent resource utilization.

To support rich object communication and scale to enormous worlds, the forwarder models object messaging between space servers as light. Each object transmits and receives communication proportional to its volume and inversely proportional to distance. Space servers assign each object pairs weight using a simple equation inspired by electromagnetic waves and uses these weights

to fairly allocate network capacity.

Leveraging geometric information and locality in this way, space servers can ensure that every object pair can communicate, even as the world grows to enormous sizes. The weight – number of photons – of an object pair may shrink very small, but it remains greater than zero. Furthermore, as the world grows, nearby objects can maintain reasonable throughput: a large number of very distant objects do not outshine a nearby one.

4.2 Computing Flow Weights

The prior sections gave an intuition for how a space server scales communication. This section describes the algorithm used. Future sections describe how it is implemented.

A space server calculates flow weight using a function $F(source, dest)$. Suppose we can define F such that as the world grows, the sum of all weights into a single server converges to a constant c. Such an F would mean that the input to a given server converges to a constant as the world's geometry grows to infinity. Furthermore, if all weights computed by F are greater than 0, then every pair will be able to communicate, as $\frac{F(a,b)}{c} > 0$. Put another way, given a server's a finite capacity, and $\frac{F(a,b)}{c} > 0$, then a pair always has a non-zero share of that capacity and can communicate.

Defining $F(source, dest)$ in terms of a simpler point pairwise function $f(p_s, p_d)$ allows us to define weights solely in terms of distance, as integrals over the region of the source R_s and destination R_d account for size:

$$F(R_s, R_d) = \int\int f(p_s, p_d) d_d d_s \quad (1)$$

One function close to the maximum bound (slowest falloff which meets the convergence and non-zero requirements) is

$$f(r) = \frac{1}{(sr + \rho)^2 \cdot log^2(sr + \rho)}$$

where s scales the falloff rate and ρ is non-zero and indirectly controls the maximum weight for an object pair.[1] In the worlds we evaluate in this paper, $s = 0.0085$ and $\rho = 0.000001$. If r is in meters, these settings mean that a 1 km region around a node receives 80-90% of its throughput and the rest of the world receives 10-20%. The r^2 falloff also means that nearby objects are guaranteed reasonable throughput (e.g., 30kbps) even when there are over ten thousand communicating object pairs.

[1]The additional log term over $\frac{1}{r^2}$ is needed because $\frac{1}{r^2}$ does not account for visibility.

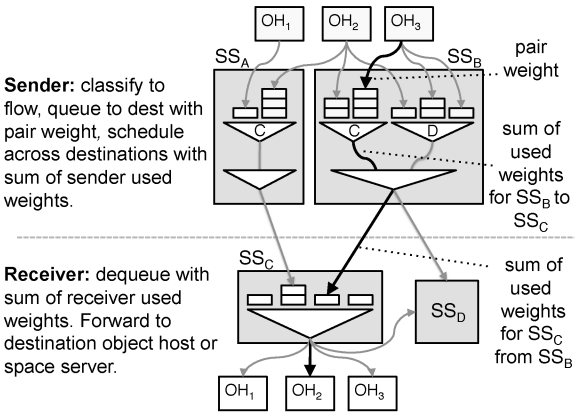


Figure 5: Fair queueing design. A packet from an object in region SS_B running on OH_2 to an object in region SS_C running on OH_3 passes through three queueing stages, shown by the dark line.

4.3 Queueing Stages

Equation 1 defines a function which assigns weights to object pairs such that every pair can communicate while ensuring nearby pairs receive significant throughput. To actually meet this communication requirement, however, the Meru forwarder needs to enforce F, such that individual object pairs receive their fair share.

To accomplish this, the Meru forwarder extends fair queueing algorithms to give object pairs capacity proportional to the weights computed by F. Logically, a Meru space server enforces fairness across all active inbound flows, as well as all active outbound flows. However, using a standard fair queue algorithm [8] to enforce fairness on both input and output is prohibitively expensive. In terms of state it requires a queue per flow, and in terms of CPU can become expensive, requiring $O(log(n))$ operations per packet, where n is the number of flows [24]. Furthermore, standard fair queueing algorithms focus on output, while Meru also requires fairly queueing input. Queueing input is a distributed problem requiring coordination between senders and receivers, while standard algorithms assume centralized information.

To reduce the costs of enforcing F with fair queues, Meru splits the task into three stages. Figure 5 shows this structure. The first stage enforces fairness within a set of flows between a pair of space servers: given an inter-server capacity C, it gives each flow its fair share of C. The second stage enforces fairness within traffic from one source server to all destination servers, fairly allocating output capacity. The third stage enforces fairness within traffic from any source server to a single destination server, fairly allocating input capacity. The first and second stages execute on the sending space server. The third stage executes on the receiving space server.

Together, these three stages follow the flow weights computed by F, no matter where a bottleneck is.

Using the algorithm described in the next section, this division allows receivers to maintain only per-server, rather than per-flow, state. Further, stages two and three have only one input per server, rather than per flow. To further reduce queueing state, Meru space servers build on core-stateless fair queueing (CSFQ) [29] for the first queueing stage. CSFQ dynamically measures per-flow arrival rates and uses these rates to calculate message drop probabilities, which allows it to drop packets without needing per-flow input queues. In our implementation, this requires 12 bytes of state per flow (as opposed to a separate queue, multiple packets in depth).

4.4 Queueing Implementation Details

This section describes the details of Meru's implementation, particularly how the queueing stages interact in a feedback loop to enforce fairness.

Space servers communicate using TCP. Under load, a Meru space server relies on flow control to explicitly signal when a source should send it messages. As space servers for a given world are under a single administrative domain (typically on the same or nearby racks), latency is very low. By configuring flow control windows to be inversely proportional to weights, a receiving space server can keep latencies low when under load.

For each flow originating from it, a space server maintains an estimate of that object pair's weight. Loc provides the source object's size and position. The server learns the destination's size from OSeg, as OSeg entries include size information (this assumes that size changes infrequently). The server approximates the destination's position as the center of the region of its authoritative space sever, learned from CSeg. It computes Equation 1 using these values. We have experimentally determined in an ns-2 simulation that when this approximation is applied to fairness on objects arranged uniformly at random, messaging each other at random, it leads to a Jain's Fairness Index of 0.96±0.03.

The first queueing stage is a core-stateless fair queue for each destination space server. Flows in the first stage have weights computed by F. Standard CSFQ assumes that the output rate is a constant (*i.e.*, the line rate). In Meru, this is not the case, as the rate at which a space server can send messages to another space server depends on the load at both servers. Meru uses dynamic measures of input and output capacity, described below, to adapt drop probabilities. Our implementation also normalizes individual flow rate estimations to the total flow rate estimation since many flows are thin, causing noisy predictors. This ensures the CSFQ algorithm does not drop more packets than it should.

Meru uses a simple feedback loop to enforce fairness in a distributed fashion. The basic approach is to compute each flow's "used weight,", u_i for flow i, which is the fraction of its weighted share it is actively using. Meru defines u_i for a flow i as

$$u_i = w_i \cdot min(\frac{r_i}{Cw_i/U}, 1)$$

where r_i is the arrival rate, w_i is the weight computed by F, C is the capacity, and U is total used weight of all active flows ($\sum u_i$). The left term computes what fraction of its share a flow is using: $0 < u_i \leq w_i$.

Used weight is necessary since weights are aggregated before being used by stages two and three. Otherwise, a high-weight flow with low utilization inflates the aggregate weight, giving other flows more than their fair share. For example, consider the case where there are two flows, with weights 1 and 100. If the weight 100 flow is only using quarter of its available share (25 units), then reporting the two flows as having a weight of 101 can lead the weight 1 flow to receive the excess capacity of 76, much more than its share. The correct aggregate "used weight" to report is 26.

Each stage one queue computes these u_i and generates a stream of packets with fairness enforced according to these u_i. The stage one queue reports a single value to stages two and three, the sum of u_i of its flows. This sum (a single floating point value) is used as the weights on the inputs of stages two and three. Because stage one has computed the used weights, enforcing fairness on the aggregate inputs at stages two and three generates the correct ratio of packets from all input streams.

U introduces a feedback loop: stages two and three compute U in terms of the u_i (indirectly via the sums provided by stage one), and the u_i are computed using U. These two computations require sending and receiving space servers to share state with one another. This state sharing is embedded in fields of inter-server messages. The sender tells the receiver the sum of used weights for its flows; the receiver tells the sender its capacity and the sum of all incoming used weights. This is a simple, cross-network control loop, as the values u_i and U are dependent on one another. However, the simplicity of the function above, combined with the minimum term, means they quickly converge.

There are several additional edge cases which we do not describe here for sake of brevity. Our experience is that making fair queueing work correctly in a real, dynamic system is much more complex than papers on the topic (and their supporting source code) would suggest.

4.5 Object Segmentation (OSeg)

A key part of the forwarding path is to determine which space server is authoritative for a message's destination.

The forwarder accesses the object segmentation service for this purpose. To improve forwarding performance and reduce load on OSeg, a space server maintains a cache of OSeg entries. This section describes OSeg's design and implementation as well the space server's caching policy. Recall from Figure 4 that space servers cooperatively invalidate stale cache entries.

OSeg is a key-value store built on top of CRAQ [31], an extension to chain replication [32] that supports both eventual and strong consistency. CRAQ is designed for read-mostly workloads, which we expect to be the case in most virtual worlds. Our measurements of Second Life show that generally only about 8% of objects move. Correspondingly, few objects need to write OSeg updates to CRAQ. To spread read and write load, OSeg organizes CRAQ nodes into a ring using consistent hashing. A particular key's replica chain are the n nodes succeeding its value. This approach allows the underlying backing store to scale with lookup, write, and storage load.

Each space server maintains an cache of recent OSeg lookups to improve message latency, reduce distortions of fairness caused by lookups, and reduce load on OSeg. Given the novel workloads virtual worlds present two major questions arise: what caching algorithm to use, and how big a cache is needed?

Intuitively, much communication in a virtual world should be local: even though distant objects are visible, interaction is mostly with nearby objects. We hypothesize that a geometric caching algorithm could be more effective than standard algorithms such as LRU. We propose a new algorithm, Lowest Forwarder Weight (LFW), based on the forwarder's weight function F. The cache evaluates an approximation of Equation 1 for each entry and evicts the lowest weight entry. Since no single source object is considered, the space server's region is used. The destination position may be approximated since locations are not always available but destination server regions always are, via coordinate segmentation (CSeg).

A space server has an LFW OSeg cache. Cache entries consist of an object identifier, server identifier, and object radius, a total of 24bytes. A small cache of about 20,000 entries occupies less than 700KB and provides low miss rates. Section 6.4 evaluates this decision.

5 Experimental Methodology

Section 6 evaluates four properties of a Meru space server: raw packet forwarding microbenchmarks, communication rate control accuracy, LFW's effectiveness in OSeg caching, and end-to-end messaging performance. This section presents the experimental methodologies we use to evaluate these properties. Because there are no well-known or well-accepted virtual world workloads, our basic experimental methodology is to use data collected from the closest analogy – Second Life – whenever possible, otherwise use models based on real world phenomena, and explore multiple possible use cases.

Our experiments focus on three variables which can strongly affect a space server's performance: object layout/movement, world size, and object messaging patterns. These variables are important because they explore how different parts of Meru's messaging system performs in different kinds of worlds with different application workloads. Object layout and movement affects messaging rates and churn in OSeg entries. World size (in geometric size and number of objects) affects the range of forwarder queueing weights and OSeg cache sizes. Finally, the traffic pattern of messages will affect the working set for the OSeg cache as well as the fraction of messages which require forwarding.

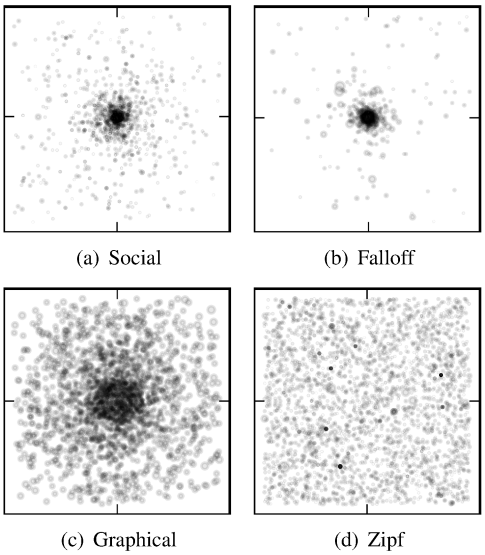
5.1 Data Collection

We draw object data from a modified Second Life client that outputs the Second Life data stream. A Second Life object is a collection of attached geometric primitives called "prims." As attachment can is also used for behavior (e.g., an avatar sitting in a chair), we infer objects by assuming that prims attached for an entire trace constitute a single object, while prims only attached for part of a trace constitute separate objects.

Second Life servers optimize the viewing experience by prioritizing updates in the view frustum. We mitigate this by continuously rotating the client. While Second Life still prioritizes updates under this workload, rotation reduces the effect and gaps between updates for moving objects on the order of one second.

5.2 World Size

To cover a representative spectrum of density we collected traces over 198 Second Life regions containing 213,000 objects at various times during March and April 2010. This translates to a density of approximately 16,000 objects/km^2. To generate a large virtual world, we create 20 million objects by copying objects from the Second Life data and uniformly distributing them over a 36 km x 36 km world, dimensions consistent with Second Life object density. A region this size covers fifteen times the area of Manhattan. We originally chose a uniform distribution to avoid motion artifacts introduced by Second Life's rigid server boundaries. We found that on a tiled version of the data caching strategies performed similarly relative to each other.



(a) Social (b) Falloff

(c) Graphical (d) Zipf

Figure 6: Query patterns for the first 3000 queries issued under the four different workloads on a 36km x 36km world. Blue circles indicate locations of queried objects, with radii showing object size.

5.3 Object Traffic Pattern

The final variable is the application traffic pattern. This data is completely inaccessible from clients and may not even map directly to our system due to abstraction breaking integration of the application into the system design. Therefore we rely on four synthetic traffic patterns in our analysis that we believe are representative of different uses of a virtual world. In each case, a source object is selected from a uniform distribution; the traffic patterns differ in destination selection. Figure 6 shows these patterns graphically when applied to a Second Life data set.

Falloff selects a destination using probabilities proportional to weights computed with Equation 1.

Social network uses a generative social network model [16] with a parameter $\alpha = 2$ to match the 90th percentile guild size of 35 in World of Warcraft [9]. The model generates a set of weighted "friends" for each object. An object picks a destination from this distribution. This captures what might be seen in a social world, where communication is mostly between friends, less between acquaintances, and rare between strangers.

Graphical selects destinations based on what an object sees. A source selects a destination object by tracing a random ray to its first intersection. This selects destinations proportional to their occluded solid angle. Due to the sparsity of Second Life, this model sees some but not much geometric locality: rays often travel far.

Content popularity mirrors the Zipfian popularity observed in web sites and peer-to-peer networks. Destinations are drawn from a probability distribution according to a Zipfian distribution. This model ignores geometry and reflects scenarios where objects popularity is independent of location. We used $\alpha = .8$ [5].

6 Evaluation

In this section we seek to answer five questions:

1. How is the space server implemented?
2. What inter-object throughput and latency can a space server sustain?
3. Does the queueing system enforce the falloff function and satisfy the communication requirement?
4. Is LFW a better OSeg caching policy than LRU?
5. Does the space server exhibit good end-to-end object messaging performance when its components are integrated?

As most virtual world systems are closed, it is difficult to measure and compare our system to them. We are able to perform simple inter-object benchmarks in Second Life, and show that our system compares favorably.

6.1 Implementation

Our current Meru space server implementation is approximately 34,000 lines of code (measured by SLOCCount [35]). It is a development fork from the open-source Sirikata platform [28] we help maintain. Currently, it does not interoperate with Sirikata's graphical client due to needed changes to communication protocols. We expect to complete re-integration within a few months, which will allow Sirikata to have worlds running on multiple space servers.

To leverage multicore processors, the space is highly multithreaded: the current implementation has eight active threads. To reduce locking overhead and simplify concurrency, each thread handles a separate set of operations and threads pass asynchronous messages. For example, message forwarding is one thread, but an OSeg cache miss passes a message to the OSeg request thread.

6.2 Microbenchmarks

We evaluate basic forwarding performance by measuring latency, forwarding rate, and throughput between a pair of object hosts. Latency measures the ping time for 64 byte messages with idle space servers. Forwarding rate measures how fast a space server can forward 64 byte messages. Throughput measures the maximum inter-object throughput using 1 kilobyte messages.

	Latency	Max Rate	Throughput
Local	692 μs	41876 pps	82.33 Mbps
Remote	1232 μs	13747 pps	47.30 Mbps
Remote Lookup	2672 μs	9454 pps	37.60 Mbps

Table 1: Space server forwarding performance.

	Latency	Throughput
Local Server	12 ms	176 kbps
Remote, distance $< 100m$	33 ms	57 kbps
Remote, distance $> 100m$	480 ms	15 kbps

Table 2: Second life message performance between two objects.

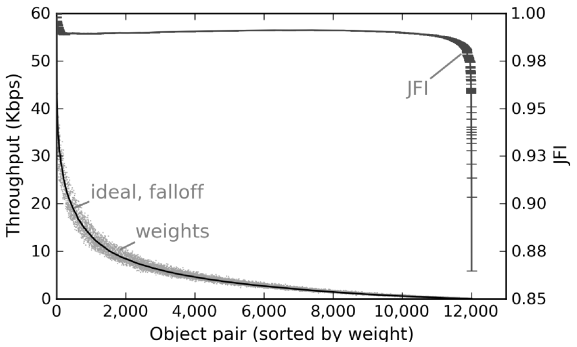
We measure three forwarding paths. In the local path, the two objects are on the same space server. In the remote path, the two objects are on different space servers and the destination is in the OSeg cache. In the remote lookup path the destination requires an OSeg lookup. Remote takes longer than local because it passes through inter-server queues, has an additional network hop, and requires a thread context switch.

Table 1 shows the results. For local messages, Meru can process over 40,000 messages per second and has a ping time below 700μs. An OSeg lookup more than doubles message latency, cuts the forwarding rate by 40% and reduces throughput by 21%: this demonstrates the need for an OSeg cache.

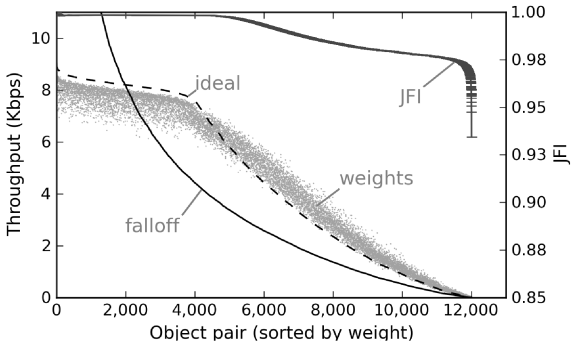
Table 2 shows results from similar experiments in a deserted and near-empty Second Life region where there are no visibility or physics computations. Local Server and proximate messaging uses llShout() and llListen(); remote uses llEmail(). Despite requiring fewer network hops since object and space simulation occur on the same server in Second Life, latency and throughput are significantly orders of magnitude worse than in Meru. These tremendous differences are due to the fact that Second Life explicitly rate-limits traffic to ensure a smooth experience. Meru's weighted rate control makes this unnecessary, permitting high utilization.

6.3 Communication Rate Control

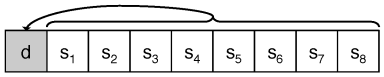
To validate whether the forwarder enforces Equation 1 and achieves good utilization, we constructed a simple linear world of nine square regions, shown in Figure 7(c). Each region is a separate space server and contains 1,500 objects. Each object in servers $s_1 - s_8$ is paired with a random object on server d. Objects have bounding volumes of 4-270 m^3 (spheres with radii of 1-8m). We evaluate two traffic patterns. In the first, objects flood the system with messages; this tests whether the space server can give all pairs some capacity while giving closer pairs reasonable throughput. In the second, objects send at a constant rate so high weight pairs use less than their fair share: this tests whether the space server can maintain high utilization. Experiments run for 15 minutes.



(a) All senders send as fast as possible



(b) High weight object pairs send less than their share.

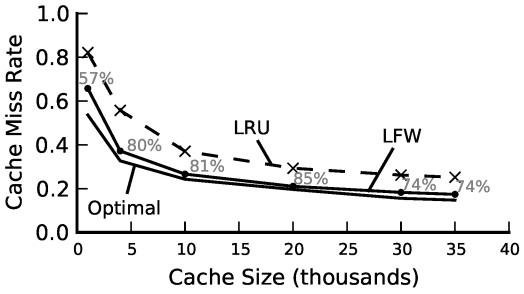


(c) Linear world topology

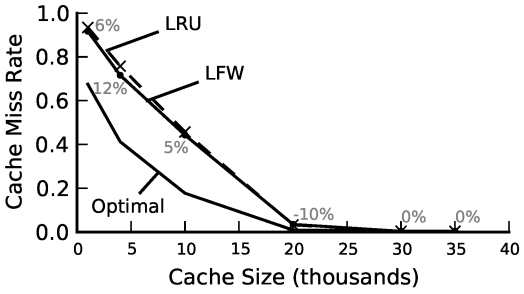
Figure 7: Flow throughput under two workloads.

Figures 7(a) and 7(b) show the results. The space servers follow Equation 1 and meet the communication requirement: nearby object pairs receive significant throughput (30 kbps), yet distant pairs can still exchange messages. Each graph shows four values for all 12,000 flows, sorted by weight. The first three show throughput (left axis): the actual received throughput, the falloff throughput (if the object used its entire fair share), and the ideal throughput (if the system enforced fairness perfectly). In the flooding experiment, the falloff and ideal are the same. The fourth value shows the JFI (right axis) of received throughput for the n highest weight pairs. The right-most data point shows the overall JFI.

The received throughput closely follows the ideal throughput. In Figure 7(a), the JFI remains high until



(a) Falloff



(b) Social, $\alpha = 2$

Figure 8: Evaluation of LFW's cache miss rate compared with LRU and an oracle

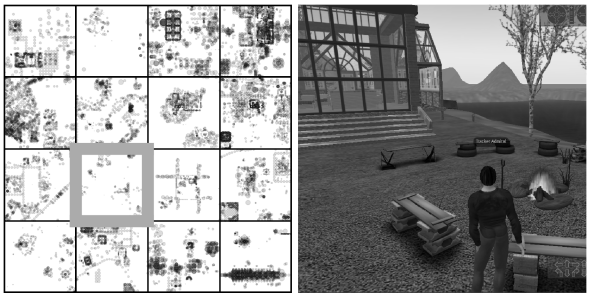
the tail: this is because in core stateless fair queueing a flow's first packet is always accepted. If the flow is for two tiny, distant objects, then under load a single packet may be much more than their share. The important point is that outside this noisy tail, JFI remains at 0.99. In Figure 7(b), JFI is much higher because space servers give the unused capacity of high weight flows to lower ones, reducing this discretization error.

6.4 OSeg

We evaluate OSeg cache algorithms by simulating the 20 million object world and traffic patterns described in Section 5. We measure the center space server as a representative cache. Objects within this server's region generate 30,000 messages per second: the portion of these that require OSeg lookups is workload dependent.

Figures 8(a) and 8(b) show cache miss rates for three cache algorithms: LRU, Lowest Falloff Weight (LFW), and an oracle optimal cache which has perfect knowledge of future requests. The percentage values show LFW's improvement over LRU as a fraction of LRU's misses over the oracle. For example, 75% means that LFW has one quarter as many additional misses as LRU.

When messaging has geometric locality, LFW can reduce cache misses by 57-85%. In the communication falloff workload, a 20,000 entry cache has a 21% miss rate. Assuming the performance results in Table 1, an LFW cache reduces the average message latency from



(a) Object layout. (b) Second Life screenshot

Figure 9: 4x4 Second Life object distribution map and Second Life screen shot captured from highlighted space server. a) Sample Second Life data for a 4x4 server grid. b) Screen shot from server in row 3 and column 2.

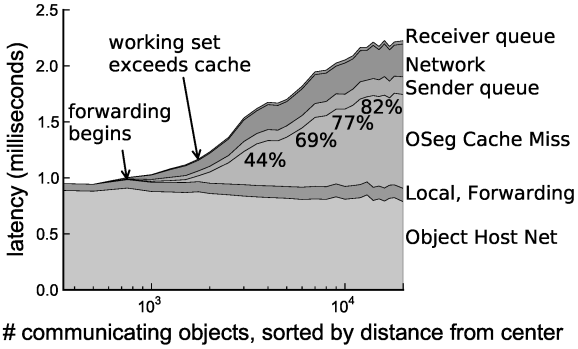


Figure 10: Average message latency in end to end experiment. Percentage show cache miss rates.

2.7ms to 1.5ms. When the working set is small, as in the social world traffic, LFW and LRU perform similarly, and a small 1MB (30,000 entry) cache suffices for a 20 million object world 15 times the size of Manhattan.

We omit data for the content and ray tracing traffic patterns because with their low locality, large datasets are effectively uncacheable. For example, with content and ray tracing traffic, a 20,000 entry cache has a miss rate of approximately 70% with the oracle algorithm. Worlds with such workloads must either optimize OSeg lookups further or have very large OSeg caches.

6.5 End-to-End Evaluation

We recreate a subset of the Second Life data described in Section 5.2, using a 16 space server grid covering a 1km^2 region with 19,000 objects, shown in Figure 9. The microbenchmarks show that messaging rate and bandwidth are largely independent of caching effects. As the purpose of these tests is to evaluate how caching and queueing interact, we focus our end-to-end tests on latency.

To measure how latency scales as the world grows,

we progressively increase the number of objects in increments of 250 every 420 seconds. Objects join the world radiating out in increasing distance from the center of the marked region in Figure 9. Senders are random samples from the marked region, and destination selection follows the social traffic pattern. The aggregate send rate is 33 one kilobyte messages per second. The OSeg cache is 256 entries to evaluate what happens when the working set exceeds the cache size.

Figure 10 shows the contributing factors to message latency as the number of objects increases. Latency increases at two major inflection points. The first occurs at 750 objects: this is when objects begin appearing on other servers and require forwarding. The "Local, Forwarding" portion of the plot begins increasing slightly as messages hit in the OSeg cache.

The second inflection point is at 1,750 objects. Network and sender queue latencies increase past this point as a greater fraction of objects are remote. At 1,750 objects, the working set exceeds the cache size, causing OSeg misses to dominate message latency. The miss rate increases until 12,000 objects, when it stabilizes at 82%. Its rate stabilizes because the social workload selects new, further objects with very low probability.

These results demonstrate the importance of the OSeg cache and its eviction algorithm to message latency. Least forwarder weight can achieve good hit rates, but even social traffic models require reasonable cache sizes. However, even when the working set exceeds the cache size or the workload is uncachable (as in graphical traffic), a space server forwards well: cache misses add only a millisecond to ping times, which remain below 3ms.

7 Related Work on Virtual Worlds

Section 4 discussed prior work that Meru builds on, such as fair queueing [8], core stateless fair queueing [29], chain replication [32], and CRAQ [31], while Section 2 described techniques worlds use to limit communication. This section presents related work on virtual worlds.

Research on how to restrict communication has examined using a radius around the object [2, 13], including orientation and recent interactions to leverage limitations of human attention [1], or partitioning the world into disjoint regions [15]. These approaches all seek to trade off user experience for increased data partitioning. Second Life, for example, does not display the world outside a user's view range. Meru takes the completely opposite approach: it allows arbitrary communication and interaction, scaling it to be more local only when under load.

PARADISE exploits perceptual limitations by aggregating uninteresting objects and providing three variable-resolution channels to support near-, mid-, and far-range viewers [27, 25]. Given Meru's communication model, we consider including such an approach a critical future step: they can, for example, aggregate thousands of tiny trees into a large forest. Exactly who in Meru's federated architecture is responsible for such aggregation remains an open research problem.

RedDwarf Server [21, 34], previously known as Darkstar, implements virtual world logic as small transactional tasks distributed across servers. Chen *et al.* [6] manage load in a virtual world by remapping highly active regions to lightly loaded servers, in a manner similar to early MMORPGs such as Asheron's Call. Another recent approach proposing pushing computation to clients, exploiting spatial properties to maintain consistency [11]. While the precise approach runs contrary to decades of experience in distributed simulation [25, 26], the core idea of federating computation is one Meru shares.

There is a rich literature within the computer graphics community that use convergent falloff functions to guarantee quality [17, 14]. Such algorithms, however, focus on narrow, centralized solutions to solve very difficult, domain-specific scalability challenges. In contrast, Meru is a distributed, federated system.

8 Conclusion

This paper presents the Meru virtual world architecture and describes the forwarding path that enables application-level messaging. Our architecture allows each component to be scaled independently. Meru takes advantage of the inherent geometry of virtual worlds to gracefully scale throughput between objects. A single pair can use the whole system capacity if available. Under heavy congestion of tens of thousands of object pairs, Meru can guarantee every pair a non-zero throughput while simultaneously giving nearby objects 30kbps. An object lookup cache can greatly reduce messaging latencies, and the geometry-aware Least Forwarder Weight (LFW) algorithm outperforms LRU significantly.

Meru is a first step towards open, federated worlds with rich application communication. Many challenges still remain before our goal of large, dense virtual worlds becomes truly practical: among them, the datastructures in PIntO to supply objects' standing queries with other important objects; the design and implementation of an addressable CDN to store object data and meshes; dynamic segmentation of the world across space servers for load-balancing; system-wide fault-tolerance; object host design; and designing an object execution environment that encourages programmability. To realize this goal, we are currently working actively on several of the research directions mentioned above and integrating them into the Meru architecture.

References

[1] BHARAMBE, A., DOUCEUR, J. R., LORCH, J. R., MOSCIBRODA, T., PANG, J., SESHAN, S., AND ZHUANG, X. Donnybrook: Enabling large-scale, high-speed, peer-to-peer games. In *Proc. ACM SIGCOMM* (Aug. 2008).

[2] BHARAMBE, A., PANG, J., AND SESHAN, S. Colyseus: a distributed architecture for online multiplayer games. In *Proc. Networked Systems Design & Implementation (NSDI)* (May 2006).

[3] `http://us.blizzard.com/en-us/company/press/pressreleases.html?081121`.

[4] BOEING, A., AND BRÄUNL, T. Evaluation of real-time physics simulation systems. In *GRAPHITE '07: Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia* (New York, NY, USA, 2007), ACM, pp. 281–288.

[5] BRESLAU, L., CAO, P., FAN, L., PHILLIPS, G., AND SHENKER, S. Web caching and zipf-like distributions: Evidence and implications. In *IEEE INFOCOM* (1998), pp. 126–134.

[6] CHEN, J., WU, B., DELAP, M., KNUTSSON, B., LU, H., AND AMZA, C. Locality aware dynamic load management for massively multiplayer games. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2005), ACM, pp. 289–300.

[7] CLAYPOOL, M., AND CLAYPOOL, K. Latency and player actions in online games. *Commun. ACM 49*, 11 (2006), 40–45.

[8] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM '89: Symposium proceedings on Communications architectures & protocols* (New York, NY, USA, 1989), ACM, pp. 1–12.

[9] DUCHENEAUT, N., YEE, N., NICKELL, E., AND MOORE, R. J. The life and death of online gaming communities: a look at guilds in world of warcraft. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems* (New York, NY, USA, 2007), ACM, pp. 839–848.

[10] `http://www.eveonline.com/ingameboard.asp?a=topic&threadID=879995`.

[11] GUPTA, N., DEMERS, A., GEHRKE, J., UNTERBRUNNER, P., AND WHITE, W. Scalability for virtual worlds. In *ICDE '09: Proceedings of the 2009 IEEE International Conference on Data Engineering* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 1311–1314.

[12] `http://highscalability.com/eve-online-architecture`.

[13] HU, S.-Y., CHEN, J.-F., AND CHEN, T.-H. Von: A scalable peer-to-peer network for virtual environments. *Network, IEEE 20*, 4 (July-Aug. 2006), 22–31.

[14] KANG, S. B., LI, Y., TONG, X., AND SHUM, H.-Y. Image-based rendering. *Found. Trends. Comput. Graph. Vis. 2*, 3 (2006), 173–258.

[15] KNUTSSON, B., LU, H., XU, W., AND HOPKINS, B. Peer-to-peer support for massively multiplayer games. In *Proc. INFOCOM* (Mar. 2004).

[16] LIBEN-NOWELL, D., NOVAK, J., KUMAR, R., RAGHAVAN, P., TOMKINS, A., AND GRAHAM, R. L. Geographic routing in social networks. *Proceedings of the National Academy of Sciences of the United States of America 102*, 33 (2005), 11623–11628.

[17] LUEBKE, D., WATSON, B., COHEN, J. D., REDDY, M., AND VARSHNEY, A. *Level of Detail for 3D Graphics*. Elsevier Science Inc., 2002.

[18] `http://nwn.blogs.com/nwn/2007/07/unwired.html`.

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

[20] PHARR, M., AND HUMPHREYS, G. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[21] `http://www.reddwarfserver.org/`.

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

[23] `http://wiki.secondlife.com/wiki/LlHTTPResponse`.

[24] SHREEDHAR, M., AND VARGHESE, G. Efficient fair queueing using deficit round-robin. *IEEE/ACM Trans. Netw. 4*, 3 (1996), 375–385.

[25] SINGHAL, S., AND CHERITON, D. Using projection aggregations to support scalability in distributed simulation. *Distributed Computing Systems, International Conference on 0* (1996), 196.

[26] SINGHAL, S., AND ZYDA, M. *Networked Virtual Environments: Design and Implementation*. 1999.

[27] SINGHAL, S. K. *Effective remote modeling in large-scale distributed simulation and visualization environments*. PhD thesis, Stanford Univ., 1997.

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

[29] STOICA, I., SHENKER, S., AND ZHANG, H. Core-stateless fair queueing: achieving approximately fair bandwidth allocations in high speed networks. *SIGCOMM Comput. Commun. Rev. 28*, 4 (1998), 118–130.

[30] STONEBRAKER, M. The case for shared nothing. *IEEE Database Engineering Bulletin 9*, 1 (1986), 4–9.

[31] TERRACE, J., AND FREEDMAN, M. J. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *Proc. USENIX Annual Technical Conference* (San Diego, CA, June 2009).

[32] VAN RENESSE, R., AND SCHNEIDER, F. B. Chain replication for supporting high throughput and availability. In *Proc. Operating Systems Design and Implementation (OSDI)* (Dec. 2004).

[33] VARVELLO, M., PICCONI, F., DIOT, C., AND BIERSACK, E. Is there life in second life? In *CoNEXT '08: Proceedings of the 2008 ACM CoNEXT Conference* (New York, NY, USA, 2008), ACM, pp. 1–12.

[34] WALDO, J. Scaling in games & virtual worlds. *ACM Queue 6*, 7 (2008), 10–16.

[35] WHEELER, D. SLOCcount. http://www.dwheeler.com/sloccount/, 2009.

[36] `http://www.wired.com/gamelife/2008/12/eve-online-offe/`.

[37] `http://www.wowwiki.com/Realms_list`.

[38] `http://www.worldofwarcraft.com/info/basics/realmtypes.html`.

[39] ZHUANG, X., BHARAMBE, A., PANG, J., AND SESHAN, S. Player dynamics in massively multiplayer online games. `http://reports-archive.adm.cs.cmu.edu/anon/2007/CMU-CS-07-158.pdf`, Oct. 2007.