SPEX: Scalable Spatial Publish/Subscribe for Distributed Virtual Worlds without Borders

Mahdi Tayarani Najaran
Computer Science Department
University of British Columbia
tayarani@cs.ubc.ca

Shun-Yun Hu
Institute of Information Science
Academia Sinica
syhu@iis.sinica.edu.tw

Norman C. Hutchinson
Computer Science Department
University of British Columbia
norm@cs.ubc.ca

ABSTRACT

Existing architectures designed to host large-scale virtual environments (VEs) use a variety of approaches, but they often limit the interaction range with other users or with the VE. How densely users can populate a given region is also limited by the hosting machine's CPU or bandwidth resources. We are motivated to remove such restrictions and present *SPEX*, an infrastructure that supports scalable spatial publish/subscribe for VE applications. SPEX is scalable and fault-tolerant, with adaptive load balancing and low latency as its key features. It is designed for the state and overlay management in VEs with many concurrent users. We evaluate a practical SPEX implementation within Amazon's EC2 Cloud and present a feasible approach to supporting 750 users across a continent with low latency, opening the possibility for hosting fast-paced games (e.g., first-person shooters) or applications on a large-scale.

Categories and Subject Descriptors

Information Systems [**Information Systems Applications**]: Multimedia Information Systems—*Massively Multiplayer Online Games*

Keywords

Cloud Game, Consistency, Scalability, First-Person Shooter (FPS)

1. INTRODUCTION

Millions of dollars are spent each year on infrastructures that host large-scale virtual environments (VEs), such as World of Warcraft [6] and Second Life [24]. The two common approaches taken to host multiple concurrent users in such VEs are *sharding* and *full partitioning*. Sharding entails running multiple parallel instances of the VE, each assigned to a different host machine. Users are then assigned to a specific shard and cannot migrate or interact across their boundaries [22]. Full partitioning, on the other hand, involves only a single instance of the VE partitioned into smaller regions, each handled independently. Users are then assigned to specific hosts based on their most recent locations and are transparently handed off to other hosts as they pass region boundaries [5].

In a sharded VE, user interactions are limited to those within the same shard. Though the VE may have a high concurrent user count (e.g., World of Warcraft has over hundreds of thousands of active users on record [29]), each user can only observe and interact with a small fraction of other users in the world. With full partitioning, the interaction range of a user is limited to a user's current region, or at most the neighbour regions when nearing borders. Long range interactions (e.g., looking through a telescope to a distant location) are fundamentally unsupported. Most systems also do not utilize resources effectively due to their static partitioning. Since all regions should be available, all servers need to be online even though the user distribution across regions may vary widely by date or time. This results in over-provisioning or under-utilization of the resources [21]. While changing the partition mapping offline alleviates the issue somewhat, it is a manual task and disrupts service uptime. Static partitioning also poses a limit on user density, as the maximum number of users in a region is capped to what a single core can simulate (with tasks such as game logic calculations, visibility management, or physics). Thus, densely crowded locations, such as stadiums or exhibits, are inherently difficult to realize (i.e., there is a *density limit* by design). Finally, First-Person Shooters (FPS) require prompt message delivery (e.g., 150ms [11]) and long-range interactions (e.g., sniper rifles). Due to the high volume of state updates per second with complicated interrelationships, large-scale VEs today cannot support FPS games without either restricting user interactions or using a centralized component.

We are motivated to remove some of these restrictions with SPEX, an infrastructure that supports scalable *spatial publish/subscribe* (SPS) [17]. SPEX is designed for VEs with hundreds of concurrent users. Users are allowed to roam freely in the VE without any density limitations while being able to interact with any user or region at arbitrary distances. The main contributions of SPEX are the following. First, we design an architecture that de-couples game logic processing from *interest management* [23], so that message routing within the VE server cluster is done very efficiently with dedicated resources. Using SPS as the primitives for interest management allows interaction ranges to be highly flexible. The SPS service is also consistent and fault tol-

MMSys '14, March 19 - 21 2014, Singapore, Singapore
Copyright 2014 ACM 978-1-4503-2705-3/14/03 ...$15.00.
http://dx.doi.org/10.1145/2557642.2557655.

erant in its stored states, by using a *distributed transaction provider* [28]. This enables us to combine publish / subscribe (pub/sub) with spatial queries, two features traditionally treated separately (i.e., pub/sub is for transient messages, while queries are done on persistent states). Second, we combine *dynamic spatial partitioning* with *scalable pub/sub* (also known as *application-layer multicast*) [10] for adaptive load balancing. This provides a two-level progressive mechanism to assign loads to different hosts, and to deliver more messages should the density level require. This eliminates the density caps of users in the VE, while balancing system operation costs. Third, by adopting a fully distributed architecture, we practically scale our implementation of SPEX to tens of hosting servers, and support 750 users with end-to-end latencies within 100 milliseconds over continental distances on the Internet.

In the remainder of this paper, we first present a background on interest management and SPS in Section 2. An overview of techniques used in SPEX is described in Section 3, and Section 4 describes how SPS is performed. Section 5 presents implementation details where use cases of SPEX can be found in Section 6. We evaluate our implementation in Section 7, compare to related work in Section 8 and conclude in Section 9.

2. BACKGROUND

A VE simulates an environment in which a group of *actors* move and interact with it based on a set of clearly defined rules, such as physical laws. Each actor encodes its behaviour in a series of *events*, and runs a local version of the simulation based on its observation of the last known states of the VE. Changes to the states of the VE are sent to the actors periodically via *update* messages. When scaling the number of actors in the VE, communicating the latest states of the VE to the actors becomes a challenge since the number of messages exchanged grows quadratically with the number of actors. *Interest management* [23] is a well-known technique to address this problem based on the observation that while the VE may be arbitrarily large, an actor has a bounded interest range, i.e., its *Area of Interest* (AoI) [27]. An actor's AoI may have any possible shape and includes all the locations in the VE of interest to the actor. Actors ensure that the VE has knowledge of their latest AoI, while the VE ensures that the actors are up-to-date on the state of the VE inside their AoI.

While the concept of AoI is widely known in VE research, there are cases where it may not fully capture the requirements of a VE. For example, while an actor's visibility or field of view can be supported by having an AoI in the VE, when a task requires making an effect on an area (e.g., casting a fire spell burning everything within a radius), the concept of Area of Effect (AoE) [25] may be more relevant. Combining both the concept of AoI (i.e., a subscription of events) and that of AoE (i.e., a publication of events) yields the more general concept of SPS.

SPS [17] is defined as the fundamental set of functions required to fully handle interest management in a VE. SPS mandates that an actor have the freedom to specify a *subscription space* and a *publication space* of arbitrary shape. Any message sent to a publication space should be delivered to all overlapping subscription spaces. From a high level perspective, most existing VE functions can be observed as special cases of SPS functions. For example, common VE operations such as chat, trade, attack, etc., can all be built by knowing neighbours spatially, or affecting other entities within a spatial radius. The significance of SPS thus is that an architecture supporting SPS can be the basis for any kind of VE. Hence, if scalable SPS becomes possible, so will a VE with a large number of concurrent users. In this context, SPEX is a novel approach to large-scale VEs by supporting SPS functions in a scalable and consistent manner.

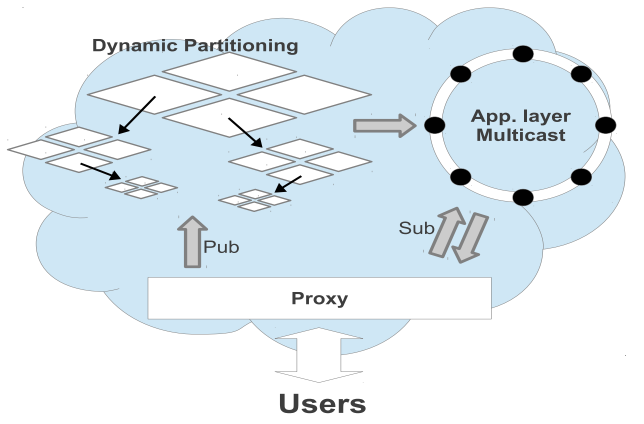


Figure 1: Basic architecture of SPEX. All communications are done using distributed transactions.

3. SPEX

3.1 Overview

Figure 1 shows the basic architecture of SPEX. We employed a cloud architecture consisting of *SPEX hosts*, a group of machines cooperating with each other to provide a unified and fully distributed service. With a cloud architecture, we are not concerned with bandwidth usage between the hosts, and CPU resources can be added on the fly. Additionally, network round-trip times (RTTs) between them are in the order of milliseconds, allowing state stored on other hosts to be retrieved in a short amount of time when required.

SPEX consists of three main components: *proxy*, *partitioning* and *multicast*. Each component is fully distributed, i.e., multiple SPEX hosts form each component. Also, a single SPEX host can be part of multiple components, e.g., be a proxy host, partitioning host and multicast host at the same time. No host ever acts as any kind of coordinator, nor is one required to maintain any form of global state.

The proxy is the frontend of the system which directly communicates with users. A proxy host serves a subset of the users and acts on their behalf in the VE. Each user sends its commands (in the form of events), such as movements and interactions in the VE, to its assigned proxy. The proxy handles all necessary functions communicating with other components and sends the required information to the user in the form of regular updates. A user has no strict affinity to a specific proxy host and can be freely handed-off from one to another. This is essential to evenly distribute users between hosts.

SPEX is designed to practically scale to tens of hosts to support hundreds or even thousands of users in a single VE. SPEX's primary goal is to overcome the limitations of sharding and full partitioning. To this end, we make no assumptions about user behaviours and impose no restrictions on how densely they can cluster. Note that this does not mean SPEX will perform under any circumstances, but will have a graceful performance degradation under heavy load rather than a complete break down. SPEX supports scalable SPS by providing a more generic API compared to traditional VE architectures. The API does not limit how users can interact with other users or parts of the VE, no matter how far apart they are.

SPS functionality is provided by using partitioning to quantize publication and subscription areas with arbitrary shapes into meaningful sub-regions of the VE. Pub/sub messages are forwarded to the hosts responsible for managing the target sub-regions. To correctly deal with potential consistency issues that could occur due to quantization and message re-ordering on different hosts, SPS functions are each executed using a separate distributed transaction which ensure correct global ordering of events across hosts. Due to unrestricted user behaviours and interactions, potential hotspots (caused by flocking) and over-congested regions are inevitable. SPEX avoids hotspots as much as possible by increasing the quantization granularity. However, when hotspots are impossible to avoid, their effects are minimized by utilizing additional resources as much as possible to multicast relevant updates. In the remainder of this section we elaborate on the details of how each task is done.

3.2 Dynamic VE Distribution

Spatial partitioning is a widely employed technique where the VE is split into a grid of fixed-sized cells, triangular strips or hexagons [19, 24, 16]. Each partition represents a VE region handled by a specific host identified by a *partition mapping*. Partitioning reduces the workload by having actors only interact with partitions that overlap their AoI, but suffers from two major limitations. First, selecting an optimal size for the partitions is challenging: a large size will result in a mismatch when mapping AoIs to regions, causing extra overhead when forwarding irrelevant events to users; while a small size will result in an excessive number of regions, inducing more load per actor operation. Second, when using a static partition mapping (e.g., computed offline using a hash function), there is a high workload variation across highly popular partitions (hotspots) and under-utilized ones.

SPEX avoids these limitations by using *progressive partitioning* combined with a *dynamic partition hierarchy*. With progressive partitioning regions are partitioned based on the density of events within them. If the activity in a region exceeds a threshold, partitioning is refined by splitting the original partition into smaller ones. Conversely, inactive partitions are merged to reduce the total number of regions. This causes the number of regions to be proportional to the distribution and frequency of events, regardless of the VE size. Figure 2 presents an illustration of a partitioned VE using both full partitioning and progressive partitioning based on a quad-tree [14]. Each dark spot represents a source of events. With full partitioning, the number of partitions is inversely proportional to the region size regardless of the event distribution. For a large VE this could result in millions of partitions. On the other hand, with the same event distribution, progressive partitioning distributes the events more evenly between fewer partitions, and avoids hotspots as much as possible.

The key enabler for SPEX's progressive partitioning is its *dynamic partition hierarchy*, which distributes partitions

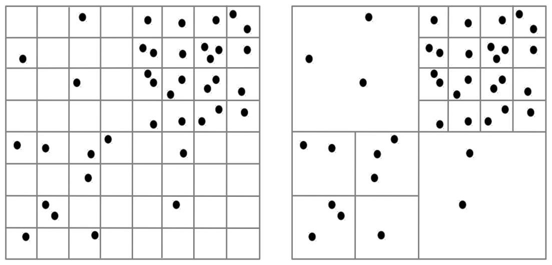


Figure 2: (left) Full partitioning vs. (right) Progressive partitioning.

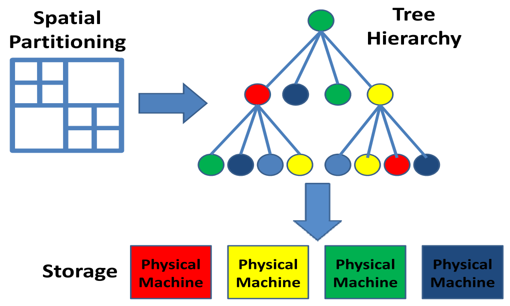


Figure 3: Distributed dynamic partition hierarchy.

between the hosts. Each partition maintains a pointer to its parent and child partitions in the form *host_id* : *partition_id*, where *host_id* uniquely identifies the IP and port of the host and *partition_id* identifies the partition. Figure 3 depicts how a VE progressively partitioned into different regions form a hierarchy, and how the hierarchy is distributed between four SPEX hosts.

To publish an event in the VE, a host finds the most suitable partitions for the publication and forwards the event to them. The root of the hierarchy encloses the entire VE and is stored in a specific host known to all others (this does not constitute a single point of failure due to replication as described later). The rest of the partitions can then be located by traversing tree links starting from the root in logarithmic steps. In practice, upper layers of the hierarchy tend to change infrequently. Thus each host optimistically caches these parts of the hierarchy, providing replication. This improves performance by reducing the communications needed among hosts when locating a partition. However, the structure of the lower parts is constantly changing. These changes are lazily propagated to hosts to avoid excessive replication overheads. If a host tries to publish an event using a stale version of the hierarchy, the publication is rejected and the host is told which parts of the hierarchy have changed. The host will then have to update its cache before retrying.

The distributed hierarchy provides a scalable way to locate partitions with low latency while not requiring any component to have a global view of the system. It is also used to answer spatial queries, e.g., finding entities along the path of a bullet.

3.3 Consistency & Fault Tolerance

An important issue that threatens the practicality of a distributed VE is *consistency*. While some inconsistencies may go unnoticed by users or only somewhat degrade its enjoyability, others endanger correctness and operability of the system. Two major sources of inconsistency exist in a distributed VE. First are inconsistencies in the VE states caused by conflicting interactions between actors. For example, if two or more actors try to perform the same operation, like picking up the same item or shooting at the same actor, there needs to be a strict and deterministic mechanism to reject all but one operation. Second are inconsistencies in the infrastructure caused by conflicting operations by the hosts due to incorrect assumptions. As an example, consider the scenario where one host is using a stale version of the partition mapping (i.e., a wrong assumption) to publish events to partitions that have previously been split into smaller ones or deleted/merged by other hosts. Such an operation should cleanly and completely be dealt with without any side effects.

Even when the assumptions are correct, a more subtle form of inconsistency arises when a single event triggers a host to send messages to multiple other hosts. Since there is no guarantee on what order each host will process the different messages it receives, inconsistencies could result due to this lack of consistent event ordering. Consider when host S_1 needs to send messages M_{S1A}, M_{S1B} to hosts A and B, respectively. At the same time, host S_2 also needs to send messages M_{S2A}, M_{S2B} to A and B. With simple message passing, there is no guarantee that A and B will both either see S_1's message before S_2's message, or vice versa. They might see them in different order, which means that they will process them in different order. Inconsistencies would show if the messages are in conflict, e.g., trying to pick up the same item as explained earlier, as each server would award the item to different players. This situation aggravates as the number of recipients increases.

We note that distributed VE systems targeting on scalability often do not explicitly deal with consistency. In the case of partitioning and sharding, interactions are limited to the current partition or shard, so consistency is implicitly ensured by design through the sole owner of the partition or shard. Other systems that try to provide a single VE instance, e.g., EveOnline [3] or Pikkotekk [4], ignore event ordering issues, but events also do not trigger message delivery to multiple servers (i.e., AoE is limited). Hence they avoid the second form of inconsistency. SPEX is designed to surpass the limitations of similar systems. Interactions are not confined to a single shard or partition, nor are AoEs restricted (via the support of generic area publications). Therefore, we carefully built in a consistency layer in the architecture of SPEX which explicitly deals with all forms of potential inconsistency.

SPEX addresses consistency by using a *distributed transaction provider*, a light-weight wrapper around the transportation layer that acts as a mediator between the hosts. It prevents inconsistencies from propagating into upper layers. A standard API for a transport layer is to provide a means to "send message M from sender S to receiver R". At some point afterwards, the message is delivered to a message handler on receiver R. The transaction provider provides a similar yet more powerful interface. First, it allows the sender to also state any assumptions they might have about the message along with it. The message is only delivered if the assumptions hold at delivery time. Second, it allows a compound message to have multiple recipients. That is, a message can consist of smaller messages, each of which is delivered to a different receiver. Finally, it ensures atomic and globally ordered delivery of the message. All recipients will receive the messages in the exact same order, or not at all. In the case of delivery, the message is delivered to the same message handlers of the receivers. The transport API offered by the transaction provider is hence transformed into "send message M from sender S to receivers R_1,...,R_n only if conditions C_1,...,C_k hold".

Using the distributed transaction provider, the actor conflict example is resolved by having an actor to send the message "pickup item only if it's still available". The host conflict is also resolved by each host publishing events like "publish event to partition only if it hasn't varied". Rejected messages are returned to the sender which should take appropriate actions. An actor failing to pick up an item knows that it should not add the item to its inventory, and the host realizes that it should update that part of its cache. However, the benefits offered by the transaction provider are not free. They come at the cost of potential extra delivery time caused by the complexity of the underlying protocol and synchronization overhead. Nonetheless, the order of extra delay is that of a network RTT, at most a few milliseconds in SPEX's architecture.

The distributed transaction provider may also provide protection against failures. Since all important communications are transmitted through it, each host may be seamlessly backed up with a number of mirror hosts. The transaction provider will ensure that backups are in perfect sync with the primary. In the event of the failure of a primary, a backup will take over and operations will resume. This obviates the need for redundant failure protection mechanisms in other components.

3.4 Adaptive Load Balancing

The load on a VE infrastructure drastically varies over time. The total number of users changes with time, and users tend to cluster in different locations based to current events [21, 22]. Having the ability to shift load between hosts is essential for both scalability and cost efficiency. Since the processing power of any given host is limited, when a host's load reaches a limit, it should be redistributed. Conversely, when the overall capacity of the hosts exceeds the total system load, loads should be shifted away from under utilized hosts so they can be safely terminated to reduce costs.

We classify load into two different components. First is distributing a large number of users between hosts. With thousands of users, tens of thousands of update messages arrive at SPEX every second, and it might stream millions of messages back. We handle this by distributing the users between proxy hosts, and use progressive partitioning to reduce the quadratic communication complexity with the number of users by forwarding events to a bounded number of relevant partitions. When a partition's load exceeds a threshold, the partition is split into smaller ones. The load balancing component ensures that the new partitions are distributed between different hosts to spread the load as evenly as possible. The dynamic partition hierarchy allows the partitions to have no affinity to a specific host, so that partitions may fluidly migrate between machines. This is especially useful when the total load of multiple partitions located on the same host exceeds its capacity, yet neither is individually high enough to be split and redistributed.

The second load component occurs when no amount of partitioning can reduce the work load to match a host's capacity. A good example is the ice rink in a hockey arena which is a single over-important hotspot that most of the actors publish or subscribe to it. Another example is the flag carrier in a capture the flag match [8]. This form of load is why partitioned VEs limit the maximum actor density in the VE. SPEX handles this by reinforcing the partition hierarchy with *scalable pub/sub*.

Scalable pub/sub (e.g., Scribe [10]) has been used for hosting distributed VEs in a myriad of different architectures [27, 19], where a *topic* (or *channel*) represents a source of events. Scalability is achieved by allowing multiple hosts to jointly disseminate a topic's messages by forming a reverse forwarding tree for each topic. This allows more hosts to help forward events to a subset of the recipients, as opposed to one host forwarding to them all. However, it comes at the cost of extra network hops per publication (another reason we employed a cloud architecture to minimize these costs). However, scalable pub/sub is not suitable for hosting a VE on its own. It provides a discrete range of topics where continuous AoIs and AoEs somehow need to be mapped to them, which is a non-trivial task. Additionally, a VE based on pub/sub alone has difficulties answering spatial queries, and inconsistencies may result from conflicting publications with nothing there to prevent them.

SPEX already handles consistency issues and has a partitioning component that maps areas to partitions. Thus scalable pub/sub fits in nicely with these components. Each partition is associated with its own topic which is created and destroyed along with it. Any events that occur in the partition are published to and disseminated by the topic. As we will see in our evaluation, adaptive load balancing plays an important role in hosting regions with densely clustered actors.

4. SPS FUNCTIONALITY

Proxy hosts receive events from users and act on their behalf in the VE. They deal with player churn and maintain enough player information to be able to translate application events into SPS functions as described next.

4.1 Spatial Publications

Publishing an event requires finding a list of partitions that intersect the area of the publication. For point publications the target will be a single partition which is the lowest one in the hierarchy (a leaf in the tree). For area publications the host must publish to the set of all leaf partitions that intersect the area. Publishing to non-leaf partitions would create parallel paths for publications, i.e., publishing to either of the ancestor partitions of a leaf. The transaction provider cannot detect the dependency between publications on different paths and would lead to inconsistencies. Thus we do not allow publications to non-leaf partitions.

Partitions are found through their parents using the partition hierarchy. Results are cached for future publications to increase locality. Once found, a single transaction is used to write the publication to the partitions. The transaction provider handles synchronizing messages on different hosts. When the transaction is executed, the event is written to the topics associated with the partitions involved in the publication, which disseminate it to other hosts interested in the event. These hosts in turn forward it to appropriate users.

4.2 Spatial Subscriptions

Proxy hosts are responsible for interest management of their users. They handle user subscriptions by maintaining a per-user subscription list, a list of all topics subscribed to by the respective user. The host then manages its own subscription list as the union of all the subscriptions lists of its users. It subscribes to each topic inside its subscription list. Once a publication is received for a topic, all subscribed local users are then forwarded the event.

The unit of subscription is a partition. Subscribing to a partition's topic is done using the partition hierarchy. A subscription list is populated by intersecting the subscription areas with partitions. This is done recursively. The subscription process starts at the root partition of the VE which covers the entire VE space. For each non-leaf partition, subscription is then refined by also subscribing to intersecting child partitions, while maintaining the subscription to current partitions. This allows hosts to have no global view of the VE but to quickly discover unknown parts as needed. As actors move in the VE their AoIs change. Modifying a subscription is done by computing the difference between the subscription areas of the old AoI and the new one then subscribing/unsubscribing to the topics accordingly.

Modifications to the partition hierarchy as a result of splitting or merging partitions are propagated to the subscribers in the form of special events. When a partition is split, the event will cause subscribers to refine their subscriptions by additionally subscribing to newly created partitions. For merge, new publications will arrive on the parent partition's topic. Recall we do not allow publications on non-leaf partitions. This is a signal that child partitions have been destroyed and the partition is now a leaf. Upon receipt, subscribers to the destroyed child partitions will unsubscribe from them.

5. IMPLEMENTATION DETAILS

5.1 Distributed Transaction Provider

Our distributed transaction provider is called *SinfoniaEx*. In SinfoniaEx, a memory pointer in the form of the tuple $\{host\_id : data\_id\}$ uniquely identifies a data item. A transaction can read or write to any data item individually, or to a group of items atomically in an all-or-nothing manner. In our current implementation we use 64-bit host ids and 128-bit data ids. Each transaction uses two-phase commit to execute in the least amount of time possible, which is 2 network RTTs. Under special circumstances where a transaction is read-only or only touches data on a single server, commit completes in a single RTT.

SinfoniaEx transactions are unlike traditional database transactions that normally allow a transaction to read and write data in multiple rounds of communication. They mandate knowledge of all data items to be read or written at commit time. Commit will run to completion unless an impossible condition is met, in which case the application is explicitly notified. Failing happens if an invalid item is referenced. Additionally, a transaction can compare a data item's value with a given value. If the two values mismatch at execution time, commit also fails. For example, assume that actor A, whose inventory is stored at $P_A = host_X : data_A$, is trying to pickup item I stored at $P_I = host_Y : data_I$. To validate that the item is still available at pick-up time, the host executes the transaction "$P_A \rightarrow$ inventory $+= I$ only if $P_I \rightarrow$ status $== AVAILABLE$".

Different transactions execute independently and are only synchronized with other potentially conflicting transactions.

This allows for higher throughput and concurrency. A conflicting transaction is one that is trying to read or write to a subset of the same data. SinfoniaEx allows synchronization in two different modes, each of which offers its own features. First is *lock mode* which uses short-lived locks during commit. Each data item referenced by a transaction is locked before reading or writing. If locks are successfully acquired, commit proceeds, else locking is retried with an exponential back-off to avoid deadlocks. Locks are held for the duration of commit (2 RTTs) and prevent race conditions due to concurrent modifications. Second is *clock mode* which uses clock vectors for synchronization [12]. During commit, the servers hosting data items of a transaction vote on the global order of executing it. They each then execute their assigned transactions in order which provides serialization. Clock mode executes lock-free and performs well under contention, but has limitations on how conditions may be expressed. Lock mode does not have limitations but lock-based systems are known to thrash under heavy contention. For the scope of this paper and our evaluation's purpose, the two modes are identical. We leave investigating the trade off between feature and performance characteristics of each mode for future work.[1]

5.2 Spatial Partitioning

Our spatial partitioning component uses Innesto [26]. Innesto is a distributed key/value store that uses spatial partitioning. We use an event's spatial coordinates as its key and the actual event as the value. Innesto supports keys with any number of dimensions so it can support a VE with any number of dimensions (e.g., 2D, 3D, etc.). We added a *publish* function to Innesto. Publish uses an event's key to compute the partitions that should receive the event and delivers it to each partition's *publication handler*.

A publication handler is hooked to the root of a topic in the scalable pub/sub component which handles pub/sub delivery. The publication handler monitors the rate of events published to its partition and decides when to split the partition. It also handles publishing special events upon structure modifications. A partition is versioned with a modification counter that is incremented each time a split or merge occurs. The version number is used to identify publications that have used stale cached entries of the distributed partition hierarchy. The publication is rejected if its provided version number mismatches the latest value. This protects the partition hierarchy from race conditions.

5.3 Scalable Pub/Sub

The scalable pub/sub component implements Scribe [10]. The hosts form a ring based on their unique host ids. Each partition serves as a different topic. The basic functionality is to subscribe/unsubscribe from different partitions. All events that occur in a partition are published to its topic. Subscription requests are routed across the ring with hops of exponentially decreasing size until they reach either the destination or an intermediate host that already has a subscription to the topic. When an event is published, the publication is forwarded along the reverse path the subscriptions came through. This will use more hosts to help forward the events, which scales as the topic becomes more popular.

[1]We made our implementation of SinfoniaEx in lock mode open-source to allow other system designers to take advantage of its numerous benefits [28].

6. DISCUSSION

One important ability of SPEX is to decouple publications from subscriptions (i.e., AoEs from AoIs). In most existing systems [24, 9, 15], the AoI of a user is tightly coupled with its position in the VE, i.e., a user publishes its location and its AoI is assumed as the area surrounding its position in the VE. SPEX allows users to fully decouple these two. A user can publish to any location within the VE, while its AoI could also be on another region in the VE. This decoupling allows SPEX-based applications to implement new features previously not possible. For example, a user could be observing a distant location through a telescope. It could also interact with the distant location by shooting at it with a long-range sniper rifle. We discuss SPEX's specific use cases as follows:

(a) Client/Server: SPEX can replace the relay/messaging server in client/server architectures, e.g., FPS games. Instead of connecting to a single server, the players connect to SPEX hosts for interest management. Additionally, server-side arbitrators could be plugged-in to subscribe to regions and handle event processing based on game logic [17] (e.g., physics). Such an architecture could be run in a dedicated environment to provide epic-scale cloud-based gaming over a WAN [27]. SPEX can also be used for LAN games, e.g., a larger scale virtual city such as the game Grand Theft Auto, where not all actors are human controlled (Non-Player Characters or NPCs) and their total number exceeds what a single server can handle.

(b) Peer-to-Peer (P2P): P2P games distribute the VE between peer (i.e., client) machines [19], each acting as a server for parts of the VE. Each peer then has to find other peers hosting events of their interest (i.e., *neighbour discovery process*), which presents a major challenge in P2P gaming. SPEX could be used as an external service to augment existing P2P architectures. Peers can simply use SPEX for neighbour discovery. They will update their states in SPEX at a low frequency [8, 27], providing a pointer to themselves (i.e., IP and port), and SPEX will find their neighbours for them. A peer will then directly contact found neighbours for higher frequency updates as it normally would in a purely P2P architecture. One could also imagine using SPEX under a hybrid architecture. For example, some peers behind NATs may have problems allowing other peers to connect. Others might not have enough networking resources to serve peers. In a hybrid architecture, a peer is given the option to either serve game states itself, only storing a self pointer in SPEX for free, or to host all their states in SPEX and let SPEX serves the peers on their behalf, an option which may require a paid subscription.

7. EVALUATION

In this section we evaluate SPEX. We emphasize that our evaluation is based on an actual implementation of SPEX running on real machines, not a simulation. We thus obtain results that are more realistic and could be of value to system developers. We use two evaluation platforms. Sections 7.1, 7.2, 7.3, 7.5 and 7.6 use the *testbed* setting: a set of 22 machines with Pentium IV 3200MHz processors and 1GB of RAM, connected with Gigabit Ethernet. Section 7.4 uses the *EC2* setting: 60 High-CPU Extra Large Amazon EC2 instances [2], each having 8 virtual cores with 2.5 EC2 compute units of processing power and 7GB of memory to repeat some tests at a larger scale with more actors. Half of the machines in each experiment are used to host SPEX and the other half act as clients. Each client machine emulates multiple actors, allowing us to evaluate SPEX with a larger number of actors than the available machines.

The purpose of our evaluation is to measure four different aspects of SPEX: 1) responsiveness by measuring end-to-end latencies of event delivery; 2) scalability by investigating the effects of varying the total number of actors on resource usage; 3) correctness by measuring *accuracy*, a metric obtained by comparing SPEX's AoI matching to that of an ideal system; and 4) stability by measuring SPEX's performance variance over long periods of time and non-uniform load.

We borrow evaluation parameters from one of the most popular large-scale VEs, Second Life [24]. Actors are randomly spread across the VE with AoIs of radius 64 units surrounding their position. An actor walks in the VE, updates its position and forwards its new position and updated AoI to its associated host. Actor positions are published as new events using point publications and each actor's subscription list is updated as an area subscription. To provide a highly dynamic fast-paced VE, we use FPS-grade update rates for the actors to move and change their AoIs, i.e., 10 moves and AoI modifications per second.

Changing the dimensions of the VE allows us to control the density of the actors. In Second Life, a server is assigned a 256×256 square region which can support at most 100 actors [5]; 16 Second Life servers can host a VE of size 1024×1024 and 64 can support 2048×2048. Using our SPEX hosts, we create VEs with these three sizes, and vary the number of actors in the VE in separate experimental runs. We investigate two types of actor behaviours, random way-points and cluster movements. In random way-point, an actor selects a random destination in the VE and moves toward it, selecting a new one once it arrives at its current destination [15, 27]. In cluster movements, the actors tend to visit certain hotspots more often than other places, causing them to cluster in the hotspots more densely [22, 17, 18]. We first start our evaluation with random way-points in Sections 7.1-7.4, then compare results to cluster movements in Section 7.5.

Each run lasts for 120 seconds [17, 18] and the VE starts as a single partition. We clip out the first 20 seconds to capture 100 seconds of run time, thus we give SPEX's load balancer 20 seconds to quickly partition the VE and balance it between the hosts. Each run is repeated 5 times to average out random actor placement effects. The duration of 120 seconds was selected to allow SPEX to stabilize, while keeping collected data traces within reasonable sizes. Nonetheless, we investigate SPEX's performance variations over a full hour in Section 7.5. Finally, Section 7.6 presents results on synchronization overheads of using distributed transactions instead of direct message passing.

7.1 Processing Latency

The quality of experience (QoE) of actors in a VE is known to be highly sensitive to end-to-end delay. End-to-end delay is the time between when an actor causes an event until it is observed by others. Previous studies have found upper bounds of 150ms for FPS games on end-to-end delay as the border of enjoyability of the game. Anything above that starts to drastically degrade QoE [11]. In a breakdown of end-to-end delay, a message has to propagate from the user machine to a SPEX proxy host (network delay), spend some time inside SPEX being duplicated and forwarded to other proxy hosts (processing delay), and another network delay back to other user machines.

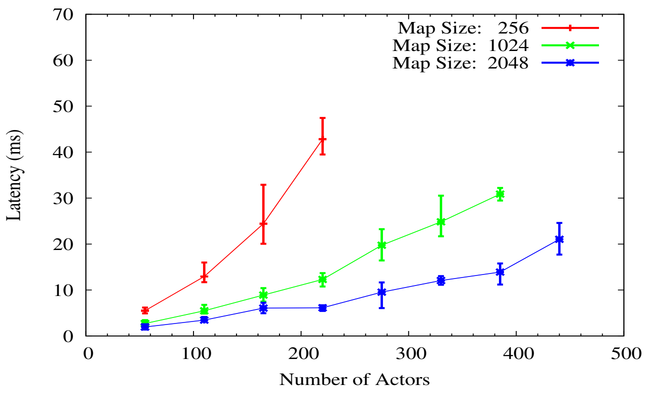


Figure 4: 99th percentile SPEX processing delays in the testbed.

We measure *end-to-end delay* as the time it takes for an event to leave one user machine and arrive at another, and measure *SPEX processing delay* as the time it takes for one SPEX host to internally deliver a publication to others. Together, both measurements allow us to quantify how the actors observe events in the VE, and how much of that is due to the distributed architecture of SPEX. If SPEX processing delay is low enough, it leaves enough time for network delays while still adhering to the bound on end-to-end delay. To obtain these measurements, we synchronize the clocks on the evaluation machines using NTP. Each event message contains two timestamps. One corresponds to the time it leaves the user machine and one is when it arrives at SPEX. The timestamps are used to calculate a message's delays. We present two characteristics of delay. Mean represents the mean delay computed for all messages in a run. We also compute an upper bound on the 99th percentile delay, measured as the 99th percentile of the maximum delays measured in 1 second intervals at each machine. The error bars represent minimum and maximum measurements across the runs of each setting.

Figure 4 presents our latency results for different VE sizes and different numbers of actors. Mean SPEX processing delays (not shown) are well under 4ms and the 99th percentile is an order of magnitude larger but still less than 50ms. As the number of actors in the VE increases, the average number of actors within the AoI of any actor increases. This puts more load on the hosts which increases processing delay.

Our latency results are significant in three important ways. First, SPEX's short mean delay allows us to use it for any type of application, even epic-scale FPS games, as it leaves ample time for network delays. Second, by using more hosts, SPEX can scale a single Second Life region to host more actors, enabling more congested areas. An interesting extension is for Second Life to keep its current partitioning method (i.e., by regular grids), but assign each partition to a separate SPEX instance rather than a single core, allowing a set of SPEX hosts to handle the interest management for a single Second Life region. Third, to host a VE of size 2048×2048 in Second Life, the operators would normally have to use 64 cores. In SPEX, the number of hosts can

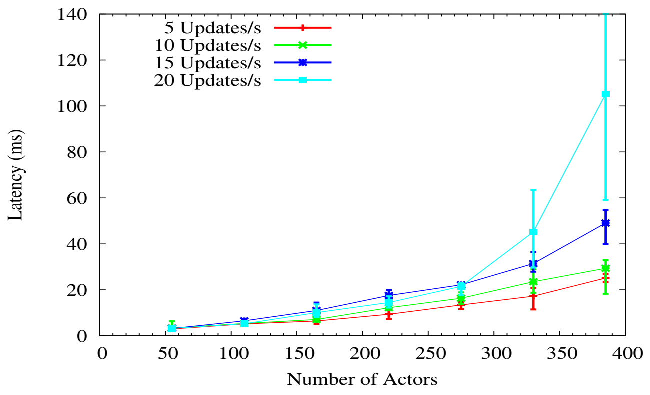


Figure 5: 99th percentile SPEX processing delay in the testbed with different update rates at map size 1024×1024**.**

dynamically scale from 1 to 64 to match current demands. Thus, an operator can balance system cost and performance with SPEX.

The per-second update frequency of each actor directly impacts performance. More updates implies that actors publish and change AoIs more frequently, putting more stress on the system to deliver more events and change subscriptions more frequently. We thus investigate the effects of update rates on maximum SPEX processing delays with a map size of 1024×1024 in Figure 5. Increasing the update rate increases the 99th percentile worst delay at larger scales. With 20 updates per second, average worst case delays are close to 100ms with a mean below 3ms. The higher update rates push the host CPU to be fully saturated, increasing delay. We expect SPEX to perform better if more hosts were available.

7.2 Bandwidth

Physical resources available on each machine are limited. We need to ensure that a host's bandwidth usage does not exceed what is available. We measure bandwidth as application throughput sent to and received by the machines. For an actor, it simply consists of the data stream uploaded to and downloaded from its associated SPEX host. A SPEX host's bandwidth consists of data it communicates with all its actors and the data it communicates with other SPEX hosts.

Figure 6(a) and 6(b) illustrate the average bandwidth measurements for each host. Bandwidth grows linearly with the number of actors, since the VE's states grow with the number of actors, and more events have to be delivered to more targets. This shows that SPEX's interest management has reduced bandwidth growth from quadratic to linear. In a more congested VE, an actor is subscribed to more events than in an uncongested VE. As evident in Figure 6(c) which presents average actor download bandwidth, with a smaller VE the number of events delivered to an actor is significantly larger than that of larger VEs. However, the required bandwidth for the actors is within the range available to residential users. As for the hosts, part of their traffic traverses the internal network (LAN). The rest is streamed over the Internet, but does not exceed bandwidth available to commercial services.

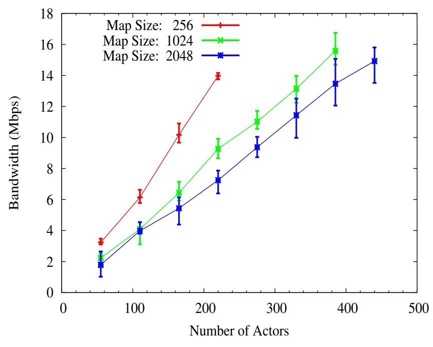
7.3 Accuracy

In this section we quantify how close the correctness of SPEX is to a hypothetical ideal system to ensure that SPEX does not trade correctness for performance. In an ideal system, all events forwarded to an actor are correctly matched to its AoI. SPEX is a fully distributed system and various delays in the system could cause forwarded events to sometimes mismatch the latest AoI. We define *accuracy* as the ratio of the number of *correct events* the actors receive divided by total events they receive. A correct event is defined to be one that would be delivered by a hypothetical ideal communication system that operates with zero delay. Events that were delivered but should not have been, and events that should have been delivered but were not, count as incorrect events. Accuracy is the only way to measure the correctness of a distributed architecture to ensure no events are lost or incorrectly disseminated in the system.

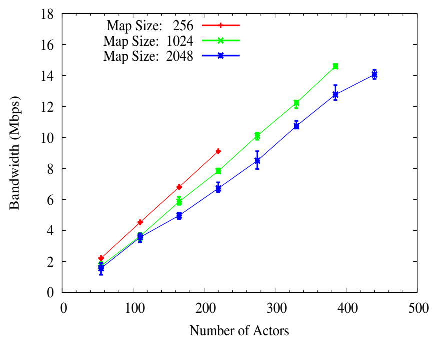
There are two sources that contribute to inaccuracy in SPEX: event processing delay and subscription delay. With processing delay, an event arrives at one host and spends some processing time going through various components before being delivered to other hosts. The longer it takes for the event to reach its destination, the higher the likelihood that the target actors will possibly change their AoIs. Thus the event delivered in an ideal system might not be delivered in a system with processing delay. When an actor makes a movement and updates its AoI, its host immediately subscribes to topics that are in the new AoI and unsubscribes from topics that are no longer in the actor's AoI. In practice, there is a delay for the pub/sub component to fix the reverse forwarding tree to include/exclude the host, something that would happen immediately in an ideal system. This is what we refer to as subscription delay. The longer it takes for subscriptions to be updated, the higher the likelihood that the actor might receive events from its old AoI but are not in its current AoI, or miss events that occurred in the new AoI before the new subscriptions take effect.

We compute server-side accuracy offline using two different timestamped logs stored by the hosts per actor: *position log*, the actor's position in the VE, and an *event log*, any event that was ever delivered to the actor. Using the position log we fully reconstruct the global state of the VE at any given time. We use this state to find the ideal set of events each actor should have received (i.e., each actor should receive relevant events within 100ms). By comparing this ideal matching to the event log we identify any missing or incorrectly forwarded events (i.e., events that should not have been received or were not received) and compute accuracy. SPEX is streaming hundreds of megabits of data per second, and for accuracy we have to log every event for the entire experiment duration, which severely affects performance. So we only compute accuracy up to 275 actors.

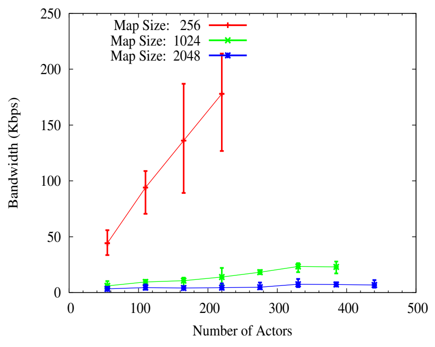
Figure 7 illustrates the accuracy of SPEX. For all VE sizes and all numbers of actors accuracy is over 90%. When the VE is large enough SPEX is over 96% accurate. Only in the extremely small VE of 256×256 does accuracy drop to 91%. Recall from Figure 4 that processing delay starts to increase in this VE size. Accuracy highly depends on system delays, as confirmed by our results. Previous studies have shown FPS games can cope with loss rates of 5%–10% as long as events are delivered within 100-150ms [7]. Compared to studies on FPS games and to related work [17, 8], we consider SPEX's accuracy to be high enough.



(a) Host upload



(b) Host download



(c) Actor download

Figure 6: SPEX host and actor average bandwidth in the testbed.

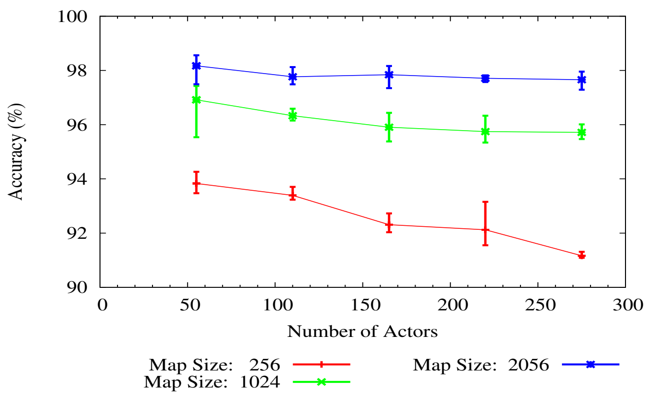


Figure 7: SPEX accuracy in the testbed.

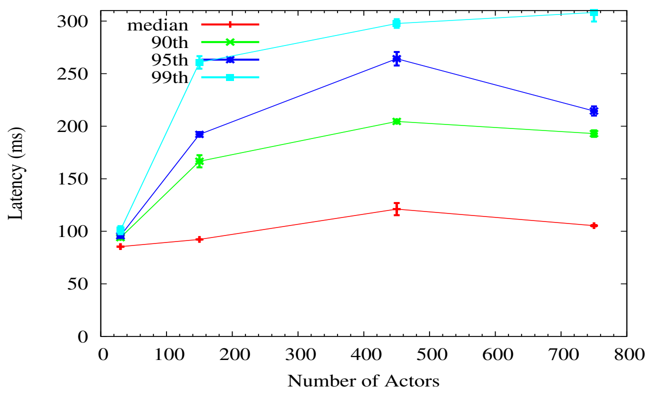


Figure 8: End-to-end delay across the continent in EC2.

7.4 Cross-Continent Latency

We now report the experiments in EC2 using almost 3 times more SPEX hosts and client machines to test SPEX for VE scales never previously tested. SPEX hosts are located in Amazon's US-EAST region (Virginia), while the client machines are located over 4000 kilometers away from the hosts in the US-WEST region (California). We found the base round-trip delay between the clients and the hosts using *ping* to be around 85ms. This distance allows us to capture the effects of communicating across a continent on end-to-end latency. Using 60 EC2 instances for multiple hours is costly, so only a VE of size 1024×1024 is used to re-run the experiments in Section 7.1 with an average of 2 runs.

At the largest scale of 750 actors, mean end-to-end delays are under 91ms, enough to host an epic-scale FPS game. Note that there is a base 85ms delay from a client to a host and back to another client, meaning update messages spend an average of less than 6ms inside SPEX. Figure 8 illustrates various end-to-end delay percentiles of the worst case delay. Parts of these worst-case delays are due to over crowded regions in the VE, where the event is being forwarded between the hosts inside SPEX. The rest are due to unpredictable events in the Internet, e.g., TCP retransmits.[2] For example, with 750 actors, the 95th percentile of worst case delay is around 215ms. From an actor's perspective, this implies that in 100 measured intervals, events over 215ms old are received only in 5 of these 100 intervals, and in the other 95 intervals all events are on time. From the host's perspective, every second, out of 750 actors, only 37.5 actors (i.e., 5% of them) receive events older than 215ms. Based on our EC2 setup, the distance between the hosts and clients is large enough to cover North and Central America, and even parts of South America. Users in these regions should expect a quality of service that is at least as good as what we present. More distant clients would observe an added delay (due to distance) compared to our results.

[2]All streams between SPEX hosts themselves and their clients use TCP to be safe to public networks, including EC2 and the Internet, respectively.

7.5 Stability

All our experiments thus far started with a single unpartitioned VE and pushed SPEX's load balancer to quickly repartition the VE and distribute it between hosts. In this section we measure the robustness of the overall architecture, and validate our experiment settings to not bias in favour of SPEX. We would like to ensure that our experiments have fully and correctly captured SPEX in its steady state operation.

We start with 275 actors in the testbed setting, the largest setting possible to fully log the system states from Section 7.3, and re-run all experiments for a full hour to measure variance in SPEX for all world sizes. If we observe significant variance in the measurements over time, our experiments would have failed to capture the steady state characteristics of SPEX. Since the hosts are streaming hundreds of megabits of data per second, it is not possible to log ev-

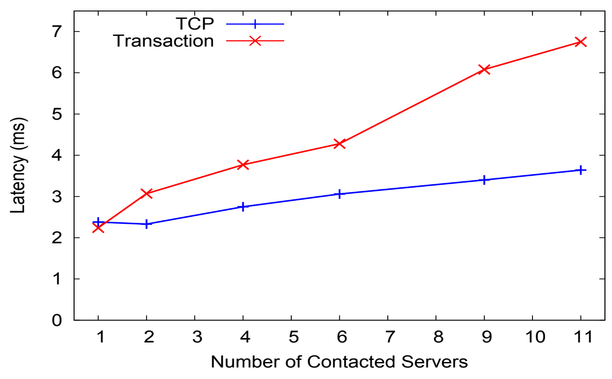


Figure 11: Cost of using distributed transactions for global synchronization instead of simple message passing.

erything for an hour. Hence, we periodically measure the system's behaviour. For latency, every 2 minutes we compute SPEX's mean and max processing delay as the average and maximum processing delay of all updates within the last 2 minutes, respectively. For accuracy, every 400 seconds we log all required information for 30 consecutive seconds, thus sampling SPEX's accuracy approximately every 7 minutes. As Figure 9 shows, we found performance samples to be consistent over time with no significant changes. Within one hour, mean delay samples varied less than 0.5ms and accuracy less than 1%. Our results here confirm that evaluations in previous subsections have correctly quantified SPEX's performance.

As a last step to validate our experimental setup, we investigate the effects of actors clustering in hotspots. In the 256×256 world size (already a congested area), we setup 1, 4 and 8 hotspots to represent locations in the VE where actors tend to investigate more often. This will cause regions around the hotspots to have a higher density of actors. With 1 hotspot, an actor is either at the hotspot and goes to another random location, or is not at the hotspot and elects to go to the hotspot 50% of the time. With 4 and 8 hotspots spread in the VE, an actor elects to go to a randomly selected hotspot with high probability (80%), otherwise it goes to a random location [17]. We ran the experiments for a full hour. As shown by Figure 10, with 1, 4 and 8 hotspots we found the mean delay to rise up to 7-8ms. Maximum delay still varied mostly between 40ms-60ms and accuracy was still maintained at around 91-92%. Hence SPEX's load balancer successfully managed actors clustering in the VE. We believe this clustering is one of the worst loads actors can impose on a VE. We expect SPEX to behave under other types of clustering (e.g., more/less hotspots, larger world size, etc.) with a response somewhat between that of uniform distribution and a single hotspot. Our results in this section validate the duration and load of our experiments, where we observe no significant change in delay or accuracy over time or over non-uniform load.

7.6 Synchronization Costs

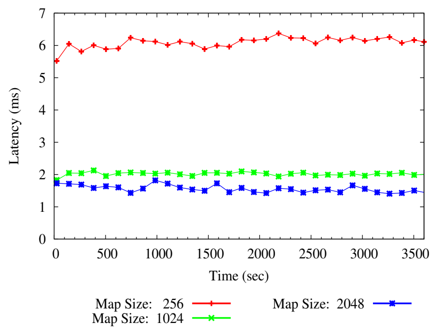
In our evaluation we have measured SPEX's latency under a workload where the actors produce point publications and have area subscriptions. We did not yet investigate area publications. This is because defining area publications, how they relate to the VE and the effects they might have on other actors is extremely application specific and cannot be easily mimicked. An area publication, depending on its size, may result in a transaction that spans multiple partitions. Since the partitions could reside on different hosts, the transaction could touch several hosts. Hence, in this section we quantify the effects of using distributed transactions for synchronizing generic events that span multiple servers. We wish to measure the basic overhead of using the distributed transaction's commit protocol to ensure global serialization. Comparison is done to the alternative of simply sending messages to servers in the form of one-to-many communications without any predictable ordering. We leave evaluating synchronization overheads with meaningful area publications as future work.

In the testbed setting, 11 machines act as servers while 5 client machines issue an aggregate of 1000 transactions per second. We vary the number of servers each transaction contacts from 1 (a normal message) to 11 (an event that effects partitions on all servers). Note that 11 represents extreme all-to-all communications. Each transaction delivers a simple 8-byte timestamp generated at commit time to its target hosts. It is designed to be small to not IO saturate the machines. A host computes the one-way delay it took for each timestamp it receives to get from the client to it, be correctly serialized according to the commit protocol, and processed by the server. We compare this delay to when the client directly sends the message to all servers. SinfoniaEx was set to operate in clock mode which performs well under heavy contention. Results are an average of 3 runs each.

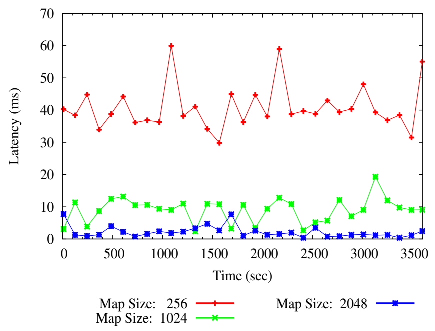
The results in Figure 11 show with 1 server, i.e., point publications or area publications that fit in a single partition, there is no distinguishable difference between using transactions or direct message passing. The commit protocol is optimized to only take a single RTT to complete for single server transactions. The number of messages that have to be delivered increases linearly with the number of servers. As a result so does latency. For higher number of contacted servers the commit protocol uses 2 RTTs. The gap between message passing and transactions increases with the number of servers but never exceeds a factor of 2x. Hence, the lower the RTT between the hosts, the lower SPEX processing delay will be. We speculate SPEX to have a higher processing delay in the presence of area publications than what we measured in Section 7.1. But as we observed in this section, the difference will be in the order of a few extra milliseconds. SPEX will live up to its promise of unrestricted area publications.

8. RELATED WORK

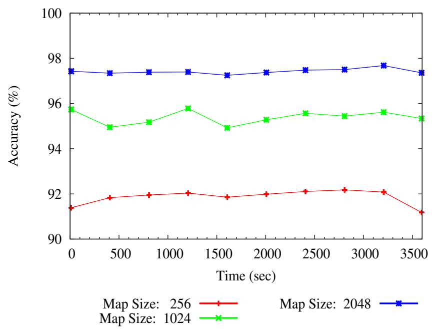
Commercial deployments Successful commercial systems currently exist that support large-scale VEs. Second Life [24] uses partitioning to assign each square region of size 256×256 to a specific core [5]. Each region can handle at most 100 actors and actors can only influence the current region they reside in. World of Warcraft [6] uses sharding to separately run multiple parallel instances of the VE [22]. Users are assigned to a specific shard and cannot migrate across them, nor can they interact with other shards. EveOnline [3] supports numerous players in a single VE. It uses a solar system architecture [1], but at its heart is a single centralized SQL database. Pikkotekk [4] has the record of supporting 999 players in single FPS game. A group of cell servers handle different *cells* (i.e., partitions) of the VE.



(a) Mean processing delay

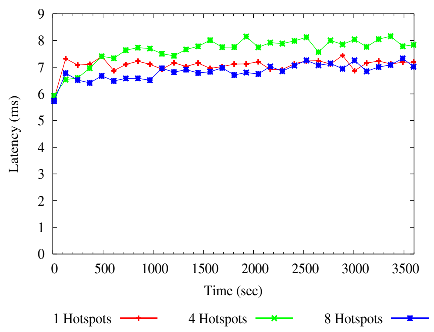


(b) Max processing delay

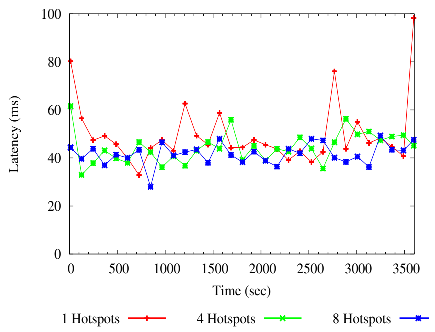


(c) Accuracy

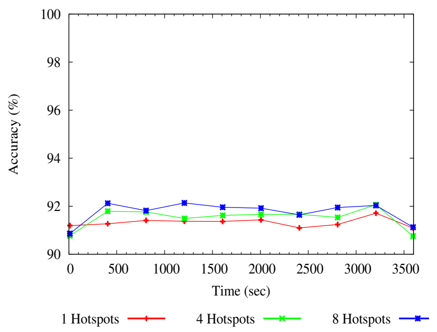
Figure 9: SPEX performance over period of 1 hour of run time with random waypoints.



(a) Mean processing delay



(b) Max processing delay



(c) Accuracy

Figure 10: SPEX performance over 1 hour of run time with VE size of 256x256 and different numbers of hotspots.

Message passing is used to communicate between the cells. However, all users connect to a central *Pikko Server* which act as a mediator between users and cell servers. While it supports area effects, distributed consistency is completely ignored. SPEX uses a fully distributed architecture. Users do not have interaction limitations, nor is there any central component. SPEX supports area publications and explicitly deals with consistency.

Query/broadcast-based Our work focuses on the support for low-latency interactions such as FPS games. Colyseus [9] and Donnybrook [8] also aim to support scalable FPS. Their approaches, as well as the later cloud-based architecture by Tayarani et. al. [27], use two classes of update rates for interest management, one low frequency position update (or spatial query for Colyseus) is used for neighbour discovery, and a high frequency update via direct connections is used for actual interactions. Donnybrook has demonstrated up to 900 players via simulations, while Tayarani et al. have shown up to 320 players interacting in a cloud environment. However, the low frequency update is broadcast globally, which imposes an inherent upper bound on scalability. The query time for neighbour discovery in Colyseus increases logarithmically with the number of actors, so the system will cease to satisfy real-time requirements beyond a certain scale. SPEX uses progressive partitioning, thus a host only needs to know about a bounded number of partitions, regardless of the total number of actors. SPEX is also practically deployed.

Partition hierarchy Spatial partitioning based systems such as N-Tree [15] and OPeN [13] both utilize quad-tree to dynamically partition the VE. N-Tree targets to support spatial multicast (i.e., publication), and OPeN focuses on spatial query (i.e., subscription). While both also provide dynamic partitioning, they do not support the combination of both publication and subscription as in SPS, and both have been mainly evaluated by simulations or analysis [15]. SPEX supports both publication and subscription, with the more general primitive of SPS, and its feasibility is demonstrated using actual experiments. Matrix [20] uses dynamic partitioning to distribute a VE between multiple hosts. However, it uses a centralized server to hold the partition mapping which is updated upon each modification, signalling a bottleneck and single point of failure. In contrast, SPEX uses a scalable and fully distributed partition hierarchy.

Pub/sub and partitioning Previous attempts have been made to use spatial partitioning in conjunction with scalable pub/sub, e.g., SimMud [19] assigns an application-layer multicast channel (based on Scribe) to each rectangular region. However, SimMud uses full static partitioning and has no load balancing. SPEX uses progressive partitioning while supporting adaptive load balancing.

SPS-based S-VON [18] and VSO [17] are two recent proposals supporting SPS based on Voronoi partitioning. S-VON enhances a Voronoi-based Overlay Network (VON), which provides spatial subscription, with the additional function of spatial publication. However, it has no built-in mechanism for dynamic load balancing. VSO provides SPS based on a super-peer design, while dynamically adjusting the Voronoi regions to balance load. SPEX differs from VSO mainly in two aspects: 1) the partitioning is based on a hierarchy as opposed to flat Voronoi; 2) a distributed transaction provider is used to maintain consistency and reliability guarantees. While VSO has demonstrated up to 1000 entities under balanced loads, both S-VON and VSO have only been verified by simulations.

9. CONCLUSIONS AND FUTURE WORK

We present SPEX, a system capable of supporting a large number of concurrent users in a VE, without common limitations, e.g., density cap or limited interactions. The system can balance scalability and operation costs by dynamically adjusting the pool of SPEX hosts. SPEX decouples publications from subscriptions, allowing users to publish to locations outside their current AoI, i.e., long-range interactions. It uses a fully distributed architecture with no components required to maintain global state, but allowing information to efficiently be located using a distributed hierarchy. Our contributions include: 1) enable long-range interactions with spatial pub/sub (SPS) based on a distributed transaction provider; 2) adaptive load balancing using a combination of progressive partitioning and scalable pub/sub to address the user density issue; and 3) evaluating the practicality of SPEX at large-scale both in a testbed and over the Internet with a real system. Our evaluation shows that SPEX scales to 750 actors and provides low end-to-end latencies and high accuracy. SPEX thus is an architecture suitable for hosting the next generation of on-line games: epic-scale FPS games.

SPEX focuses on VE distribution, interest management and message dissemination. Distributed transactions provide the means to correctly synchronize events spanning multiple servers, but in this paper we did not extensively deal with conflicts occurring between users, which is important to support more sophisticated game states and logic. Our future work is thus to create a complimentary component to simplify the dealing of user conflicts. SPEX takes user events and translates them to SPS functions. It creates per-user interest sets and AoEs. A future system could use this information and distributed transactions to deal with user interactions and conflicts. By using multiple distributed transactions per user to over estimate users' range of effects, we can then deterministically resolve conflicts in a fair manner.

10. REFERENCES

[1] `http://www.gamasutra.com/view/feature/132563/infinite_space_an_argument_for_.php?page=1`.

[2] Amazon ec2. aws.amazon.com/ec2.

[3] Eve online. http://www.eveonline.com.

[4] Muchdifferent's pikkotekk. www.muchdifferent.com/?page=game-pikkotekk.

[5] Second life server architecture. wiki.secondlife.com/wiki/Server_architecture.

[6] World of warcraft. http://us.battle.net/wow/en/.

[7] T. Beigbeder et al. The effects of loss and latency on user performance in unreal tournament 2003. In *NetGames '04*, pages 144–151, 2004.

[8] A. Bharambe et al. Donnybrook: enabling large-scale, high-speed, peer-to-peer games. In *SIGCOMM '08*, pages 389–400, 2008.

[9] A. Bharambe, J. Pang, and S. Seshan. Colyseus: a distributed architecture for online multiplayer games. In *NSDI'06*, pages 12–12, 2006.

[10] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 2002.

[11] M. Claypool and K. Claypool. Latency and player actions in online games. *Commun. ACM.*

[12] J. Cowling and B. Liskov. Granola: low-overhead distributed transaction coordination. USENIX ATC'12, pages 21–21, 2012.

[13] S. Douglas et al. Enabling massively multi-player online gaming applications on a p2p architecture. In *Proc. Information and Automation*, 2005.

[14] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

[15] C. GauthierDickey, V. Lo, and D. Zappala. Using n-trees for scalable event ordering in peer-to-peer games. In *Proc. NOSSDAV.* ACM, 2005.

[16] S.-Y. Hu, S.-C. Chang, and J.-R. Jiang. Voronoi state management for peer-to-peer massively multiplayer online games. In *Proc. IEEE CCNC*, 2008.

[17] S.-Y. Hu and K.-T. Chen. Vso: Self-organizing spatial publish subscribe. In *Proceedings of IEEE SASO 2011.*

[18] S.-Y. Hu et al. A spatial publish subscribe overlay for massively multiuser virtual environments. In *ICEIE 2010*, 2010.

[19] B. Knutsson et al. Peer-to-peer support for massively multiplayer games. In *INFOCOM 2004*, mar. 2004.

[20] R. Krishna Balan, M. Ebling, P. Castro, and A. Misra. Matrix: adaptive middleware for distributed multiplayer games. Middleware '05, 2005.

[21] Y.-T. Lee and K.-T. Chen. Is server consolidation beneficial to mmorpg? a case study of world of warcraft. In *Proc. IEEE Cloud*, 2010.

[22] J. L. Miller and J. Crowcroft. Avatar movement in world of warcraft battlegrounds. NetGames '09, 2009.

[23] K. L. Morse, u. Bic, and M. D. Dillencourt. Interest management in large-scale virtual environments.

[24] P. Rosedale and C. Ondrejka. Enabling player-created online worlds with grid computing and streaming, 2003.

[25] R. Sueselbeck et al. Adaptive update propagation for low-latency massively multi-user virtual environments. In *Proc. ICCCN 2009.* IEEE, 2009.

[26] M. Tayarani Najaran and N. C. Hutchinson. Innesto: A searchable key/value store for highly dimensional data. In *CloudCom '13*, 2013.

[27] M. Tayarani Najaran and C. Krasic. Scaling online games with adaptive interest management in the cloud. NetGames '10, pages 9:1–9:6.

[28] M. Tayarani Najaran and C. Krasic. SinfoniaEx : Fault-Tolerant Distributed Transactional Memory. Technical report, 2011.

[29] J. Waldo. Scaling in games & virtual worlds. *ACM Queue*, 51(8), 2008.