VSO: Self-organizing Spatial Publish Subscribe

Shun-Yun Hu and Kuan-Ta Chen
Institute of Information Science
Academia Sinica
Taiwan, R.O.C.
Email: {syhu, swc}@iis.sinica.edu.tw

***Abstract*—Spatial publish subscribe (SPS) is a basic primitive underlying many real-time, interactive applications such as online games or discrete-time simulations. Supporting SPS on a large-scale, however, requires sufficient resources and proper load distribution among the simulation units. For load distribution, existing mechanisms either use a static partitioning, such that over-provisioning or overloading are bound to occur, or require manual adjustments unsuitable for massive workloads.**

We describe Voronoi Self-organizing Overlay (VSO) [1], which extends a Voronoi-based Overlay network (VON) to automatically partition and manage a logical space to support SPS. Efficient resource usage thus is possible as only the units necessary to maintain the system are used. Load is also balanced among the resource units so that overloading or over-provisioning can be avoided. We use simulations to verify our design and describe some preliminary results.

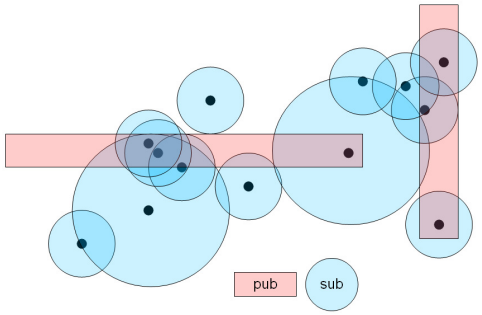


Figure 1. Schematic for a 2D Spatial Publish Subscribe (SPS). Messages are delivered between entities only if the publication areas (e.g., rectangles) overlap with the subscription areas (e.g., circles). [7]

I. Introduction

Spatial simulations are computing environments that allow independent *entities* with coordinates to move within a coordinate space, according to certain rules and the progression of a logical time. From the rigid-body and molecular dynamics (MD) simulations in physics, to discrete-time military simulations, to virtual worlds (e.g., Massively Multiplayer Online Games, or MMOGs), examples of spatial simulations abound and are important to provide understanding or interactions in otherwise expensive or impossible scenarios.

Underlying various simulations is a common requirement to know other entities within a certain range (referred hereafter as the *area of interest*, or AOI of an entity), and the ability to exchange messages with them. For example, in a MMOG, players need to know their *AOI neighbors* (i.e., other players within visibility [2]) so that the positions and actions of these neighbors can be properly displayed.

Such requirements can be supported by either a *spatial query* (i.e., finding all entities within a space) [3], [4] or a *spatial multicast* (i.e., sending messages to all entities within a space) [5], [6]. Although functionally similar, each has its own merits and limitations. For example, in a military simulation, both a radar and a foot soldier might detect an incoming tank, though the radar likely has a longer sensing range. To support this scenario, we could either use spatial multicast to transmit the positions (and action events) of each entities, or use spatial query periodically to find the entities of interest. However, if we transmit the tank's position via spatial multicast alone, finding a delivery range suitable to both radars and soldiers can be non-trivial (e.g., a long range for radars may be overkill for soldiers, while a short range for soldiers may not be enough for radars). It is thus more reasonable to let the radar and soldier perform spatial query periodically for nearby entities, in order to properly see the tank. However, if the tank fires a missile that can hit a far away target, a potential target may not necessarily have the right sensing/query range to detect and learn of the tank's firing action (e.g., an artillery whose sensing range is smaller than the missile's firing range, will not learn about the tank and its firing action). Although we can also let the missile to perform spatial query constantly to find entities it might hit (and then notify the entities directly), such "query then notify" introduces longer latencies than direct notifications, which may not be desirable in latency-sensitive applications. In this case, a spatial multicast is a better option, as the event of the fired missile can simply be delivered along the firing path to reach potential targets.

We suggest that both spatial query and spatial multicast can be combined into a more general *spatial publish subscribe* (SPS) [7] mechanism, where each entity indicates its interest to receive updates (i.e., to subscribe) from a specified space within the coordinate system, and sends messages (i.e., to publish) to a specified space. An entity will only receive a published message, if its subscription space overlaps with the publication space of a message (see Fig. 1). The simplest approach to support SPS is to let each

entity send its position updates to a central manager, and have the manager to match and filter only relevant messages back to each entity (i.e., performing *interest management* [8] for each entity). However, the total amount of resources of this manager determine the scale of the system. To perform scalable SPS, interest matching needs to be divided and assigned to different simulation units. To limit our discussions, we will focus on discrete-time simulations with strict latency requirements in 2D space, such as MMOGs, as its requirements are one of the strongest. But we note that our discussions can be generalized to higher dimensions and other types of spatial simulations, especially those with more relaxed accuracy or latency requirements.

Various techniques have been proposed to divide the workload for spatial query or spatial multicast, from divisions based on space or on entities (see [9] for a survey). However, to the best of our knowledge, few approaches have been described in literature on the actual design and evaluation for the full SPS functionality. While the idea of SPS has also been described by the Data Distribution Management (DDM) section of the IEEE 1516 simulation specification (the High Level Architecture, or HLA) [10], the specification intentionally leaves out the implementation details. In this paper, we describe Voronoi Self-organizing Overlay (VSO), a spatial division approach to support SPS based on Voronoi partitioning and self-organizing load balancing. The overall effect is that only the resource units necessary to support the system actually join the system, thus avoiding the over-provisioning or over-loading issues commonly associated with static partitioning.

II. Background

A. SPS

Traditionally, the most basic form of publish/subscribe (pub/sub) is channel-based (also called topic-based) [4], [11], where receivers would subscribe to a particular channel for messages sent by publishers to the channel. However, in case for spatial messages, how to divide the space and assign them to channels becomes problematic (i.e., how to determine the right region size [12]). Spatial publish / subscribe (SPS) on the other hand, provides a simpler concept for these operations. The simplest SPS supports the subscriptions and publications of messages on an area. For simplicity, we will discuss 2D space only in this paper. As the smallest area is one with zero size (a point), SPS can be also be specified with points. There are thus four main types of SPS operations: 1) *area subscription* specifies an intention to receive all messages published onto the area specified; 2) *area publication* sends a message to an area, receivable by any subscribers whose subscribed areas overlap with the publication areas; 3) *point subscription* intends to receive any area publications that cover the point specified; 4) *point publication* sends a message to a given point, receivable by any area subscribers whose subscribed areas cover the point.

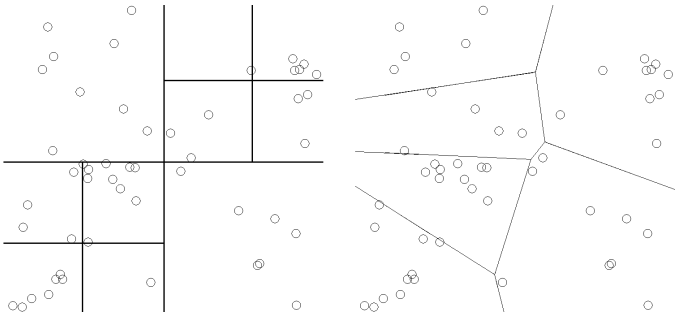


Figure 2. Quad-tree (left) vs. Voronoi (right) Partitioning.

Spatial partitioning is easier to understand and implement, and requires less application-specific knowledge to generalize its use. Existing spatial partitioning methods, however, either divides the virtual space in pre-defined manners (e.g., division by grids) which makes picking the right region size difficult [12], or requires human intervention in specifying the different region sizes. When the entity movement or density changes in highly unexpected manners, or if a large number of entities are simulated, such approaches become unpractical and unsuitable. Ideally, if spatial divisions can occur based on the density and distribution of the entities, and adjust automatically as the density changes, then we will have a system that is more flexible to deploy, and more practical to manage. In this paper, such an approach is described based on Voronoi partitioning [13], as it allows fewer divisions of the space than alternative approaches, especially when the entities' spatial distribution is skewed. For example, in Fig. 2, we see that for 50 entities around a few clusters, if a region can accommodate at most 10 entities, a quad-tree-based partitioning would produce about 10 regions, while a Voronoi partitioning produces only 6. Although concepts of dynamic zoning have been proposed before (e.g., Matrix [14]), when the management of entities is distributed, issues of state consistency, load balancing, and fault tolerance emerge and need to be considered in whole. We thus also describe our techniques to address these issues, which existing proposals have yet described in a comprehensive manner.

B. Multi-user Virtual Environments

Recently, MMOGs have become a popular form of entertainment that rivals with movies. In a MMOG, users send *events* (e.g., movements, trading, attacks) to a game state manager, which, according to current game states and the rules of the game (i.e., *game logic*), will update the game states and return *updates* to the users, within their AOI. We thus can consider sending events as point publications, made by users and subscribed by the state managers, who have well-defined subscribed areas. In the case of a partitioned (i.e., zoned) world, each manager can subscribe to non-overlapping regions. Sending updates from managers to

users can also be seen as point publications of game state updates, receivable by users that have previously performed area subscriptions in their AOIs. This abstraction is general enough that if scalable SPS systems exist, MMOGs may scale up accordingly.

C. Voronoi Overlays

Voronoi diagram [13] is a way to divide a space into n regions based on the positions of n points, or *sites*, such that all points within a region are closest to that region's site than to any other site. Intuitively, we can see each region in a Voronoi partition as the sphere of influence of a given site, or that each site is the closest manager of any point within a region. By adjusting site positions, we can shift the *edges* of a Voronoi diagram to change a region's shape and size.

Existing works describe how Voronoi can provide decentralized node management. For example, Liebeherr *et al.* [15] describe how to build a Delaunay Triangulation (DT) overlay (i.e., a dual structure of Voronoi) in a distributed way to provide routing from any node to any other node efficiently. However, it is a static overlay where the nodes are relatively static in positions (e.g., only insertion and deletion of nodes are considered). On the other hand, a Voronoi-based Overlay Network (VON) [13] is designed to accommodate nodes that move their positions constantly. Each node is always aware of a number of AOI neighbors within a radius. By using a Voronoi diagram to organize the AOI neighbors, each node can identify the *boundary neighbors* (i.e., AOI neighbors whose Voronoi regions overlap with the AOI boundary, see Fig. 3 left), and enlist their help in notifying any potential neighbors. New AOI neighbors thus can be discovered without relying on a central server (see Fig. 3 right, where AOI neighbors are updated).

To spread message to all nodes within an AOI (i.e., an *AOI-cast*), a unique spanning tree can be constructed on top of a Voronoi overlay. VoroCast [16] constructs such a spanning tree in a distributed fashion and covers all AOI nodes only once (see Fig. 4). Genovali and Ricci [17] have also shown that while only nodes inside the AOI need to involve for circular AOIs, when the AOI is rectangular, constructing the spanning tree may need to involve other nodes outside the AOI.

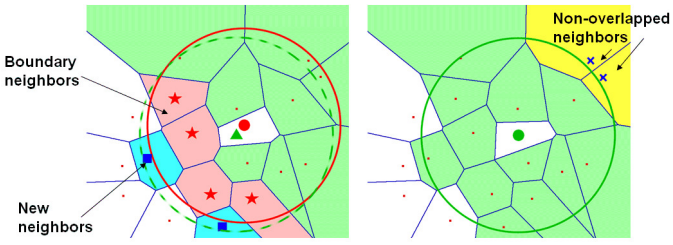


Figure 3. Neighbor Discovery in VON. Big circle is the AOI boundary. When the circle node moves to the triangle, stars are its boundary neighbors, and squares are new neighbors. Crosses are old neighbors to disconnect.

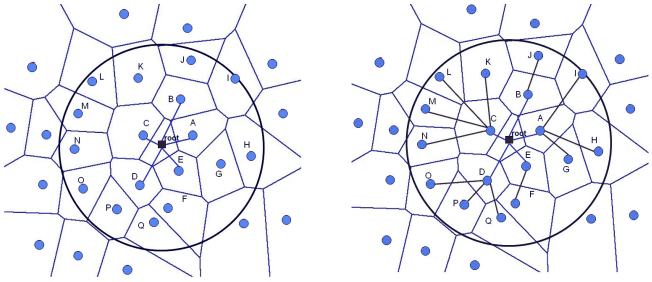


Figure 4. Non-redundant spanning tree to forward messages in an AOI.

III. Voronoi Self-organization

Voronoi Self-organizing Overlay, or VSO, follows the basic idea of spatially partitioning a virtual space to support SPS. The entire space is divided into various *regions*, each managed by an *interest matcher*, or simply *matcher*. The matcher is responsible to map a given publication with potential subscribers. We define a *client* as a participant (i.e., entity) of the system that may perform publications or subscriptions. Additionally, clients can *move* their subscribed areas, to receive more relevant and timely update messages. For simplicity, we assume that these pub/sub areas are circles or rectangles, and have a well-specified *center* point (e.g., a circle is a center plus a radius; a rectangle is a center plus a width and height). Each *subscription* thus is defined by a subscription area, and each *publication* has the form of $(area, message)$, where $area$ is the publication area, and $message$ is an application-specific message. Each matcher is the unique authority within the region, such that a client needs to register its subscription interests with at least one matcher, before it can receive any publications. To ensure that each subscription is only handled by a single matcher, the matcher whose region covers the center point of a subscription, is the proper *owner* of the subscription.

The system starts when clients contact their owner matchers and specify subscription requests. If the request is sent to a non-owner matcher (i.e., the matcher's region does not cover the subscription point), the request is forwarded greedily based on the subscription center to the actual owner matcher. Note that greedy forward generally takes $O(N^{1/d})$ time on a Voronoi overlay (where N is the node size and d is the dimension), though it can be improved to $O(log(N))$ by adding *shortcuts* [18]. As such, bootstrapping can be done by simply contacting any one of the existing matchers. The first matcher of the system is called the *gateway*, and may be a well-known host to facilitate client join. Once in the system, clients can move subscription centers to new locations, thus changing the subscribed areas. If the subscription center crosses the boundary of a matcher into another matcher's region, then an explicit *ownership transfer* occurs where the old and new owner exchange messages to ensure that the transfer is atomic.

When a publication occurs, it is first sent from a client to its owner matcher, which then checks a list of known subscribers to send the message. If the publication area falls outside the owner matcher's region, the publication will be forwarded to neighboring matchers continuously, until all affected matchers are notified (via forwarding such as VoroCast [16]). This way, each publication is guaranteed to be delivered to all potential subscribers, as long as subscription info is properly maintained by each matchers. Although a subscription area may span several regions, each subscription has only one owner, who is responsible to forward any relevant publications to the client. A published message thus would be delivered to a subscriber only once.

A. Dynamic Load Balancing

As subscriptions move and cluster, a given matcher may become overloaded, it is thus desirable to adjust matcher loads automatically. For this purpose, the matchers form a Voronoi-based Overlay Network (VON) [13] to maintain connectivity and perform load balancing. While a VON provides neighbor discovery in a distributed way, functionally it can be seen to provide only area subscriptions and point publications. There is also no load balancing in VON's original design. By alleviating a VON to handle only matcher connectivity (instead of clients), a client in VSO maintains only a single connection with its matcher. To provide load balancing, the matchers can adopt two main strategies to alleviate overloading:

1) Adjust region size to match workload with capacity;
2) Insert a new matcher nearby to share the loads.

For the following discussion, we first make two assumptions: 1) a matcher can detect whether it is overloaded (e.g., simple CPU or memory usage monitoring); and 2) potential matchers can identify themselves and report availabilities to the gateway, which acts as the initial entry point of the system for all candidate matchers. The gateway may send a message to promote a matcher candidate at a certain location in the virtual space. A potential matcher may be a provisioned server (i.e., in a cloud system), or a client machine (i.e., in a P2P system), however, this is mainly a deployment decision and not architectural.

We note that in a Voronoi partitioning, it is fairly easy to adjust region boundary, by moving the *sites* of the Voronoi diagram (see Fig 5). The main questions then are: 1) how to move the site locations, as client density changes; and 2) where and when to insert new matchers, if overload persists.

We experimented with a number of approaches and came out with the following simple rules, which we found were effective. Intuitively, the ideal partitioning makes the size of the region to match the region's loading, which should reflect the capacity of the matcher. In other words, the region size would change in such a way that the average client density per square area roughly equals the handling ability of the matcher per square area.

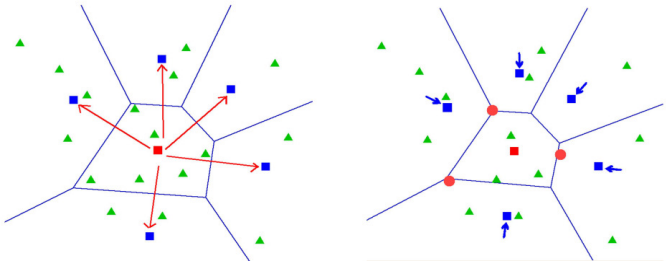


Figure 5. Schematic of load balancing. (Left) The center matcher requests help from its neighbors. Triangles are the subscriptions centers of clients. (Right) Neighbor matchers move their sites closer to make the center region smaller. Note that some subscription centers are now in neighbor regions, the ownerships thus are transferred (e.g., the circles at the boundaries).

In our rules to adjust region sizes, a given matcher will:

1) shrink region size when overloaded by asking neighbors to move their sites closer.
2) request a matcher insertion from the gateway, if 1) does not work after a while.

The threshold to adjust would require tuning in practice, we thus set the number of the "come closer" requests as an adjustable VSO_INSERTION_TRIGGER, so that an overloaded matcher's neighbors will attempt to move their sites closer for a specified number of times, before the gateway is called to insert a new matcher. A neighbor matcher uses the following formula to move closer:

$$P_{new} = P_{curr} + (P_{overload} - P_{curr}) * F_{move} \quad (1)$$

Where P_{new} is the new position to be taken by the neighbor; P_{curr} is the current position of the neighbor; $P_{overload}$ is the overloaded matcher's position and F_{move} is a number between 0 and 1 for the speed of adjustment. While the above rules may appear intuitive, the site positions would soon come close to one another during adjustments, thus making the clients to cross region boundaries easily. A second requirement thus is to have each region centers (e.g., the *sites* of the Voronoi diagram) to be as far away from each other as possible, while ensuring that each region can approximately contain a cluster of clients, to minimize region crossing. We thus devise a third rule as follows:

3) a matcher will move its site location closer to the center of clients, using the following formula:

$$P_{new} = P_{curr} + (L_{center} - P_{curr}) * F_{adjust} \quad (2)$$

Where P_{new} is the new site position of the matcher, P_{curr} is the current site position of the matcher, L_{center}

is the *load center* of the client positions (defined as the average coordinates of all x and y coordinates of their subscription centers), and F_{adjust} is a parameter between 0 and 1 for how much should be adjusted each time. Note that the matcher's final site position will be a composite of both formula's effects.

Matchers can depart from the system, if its load is below a given threshold, so that less overall resources are used to support for the same workload. However, for simplicity, we currently designate that only when the workload is zero (i.e., there is no subscription records on a particular matcher), will the matcher leave the system.

An important note about dynamic load balancing is that the change in site positions are coordinated via VON's Move Procedure [13], and are only loosely synchronized. The change in site positions need not be coordinated atomically among relevant neighbors, because even if temporarily inconsistency in the knowledge of site positions exists, it will become eventually consistent when the load behavior stabilizes (i.e., when load adjustment is not needed).

B. Cross-Boundary Interactions

With the division of the virtual space into regions, the subscription of a client may fall into more than one matchers. We thus need to ensure that subscriptions across region boundaries can still work correctly. VSO uses two basic designs for this purpose:

1) Single owner To ensure the consistency of the subscription (i.e, the client's understanding is the same as the matcher's), each subscription is maintained authoritatively by one matcher only, and ownership of the subscription is transferred via explicit messaging. As clients move (i.e, change their subscribed area), the subscription information must also move to a new matcher. Here an explicit ownership transfer is used, so if a subscription's center has moved into the region of a neighboring matcher over a certain time period (defined by an adjustable TIMEOUT_TRANSFER), an ownership transfer message is sent from the previous to the new owner matcher. The new owner would send back an acknowledgment for the transfer. If the ownership transfer is unacknowledged for a certain period of time, the previous owner may reclaim the ownership of the subscription.

2) Soft-state replications To ensure that subscribers can still receive updates from matchers in other regions, a subscription is replicated at all other non-owner matchers if the subscription overlaps their regions, and is maintained with a soft-state model (i.e., temporary inconsistency is allowed to occur between the primary and replica, but will eventually become consistent if the primary subscription is not continuously changing). The matchers with the *replica subscriptions* can thus check if any publications occurred within their regions need to be delivered to a subscriber at a nearby region. Such publications will be forwarded to the owner matcher of the subscriber first, so that the owner matcher can learn of such publications and notify the subscriber only once for each publication. Note that such notifications are necessary because a publication area may overlap with a subscription area at a region different from the one of the subscriber's owner matcher (see Fig. 6).

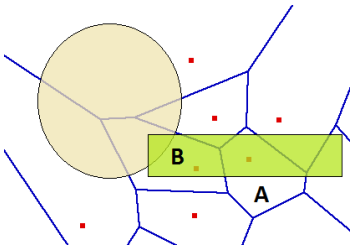


Figure 6. A publication (circle) may overlap with a subscription (rectangle) area at a different region (i.e., matcher B) than the subscription's owner matcher's (i.e., matcher A)

C. Fault Tolerance

When regions are divided and managed by separate matchers, one natural issue is how the publication / subscription will still work in spite of matcher or client failure. Client failure is simple to resolve, where the matcher that currently handles the client can detect a connection loss and remove the client's subscription. For matcher failures, we use two mechanisms to handle the failure:

1) Besides the owner matcher, each subscription is also backed up at one or more neighboring matchers. If the subscription area overlaps the regions of other matchers, those matchers are notified of the subscription, plus any future updates in the subscription (e.g., when the subscription moves). Even if the subscription area does not overlap with other regions, it is minimally backed up at the closest neighboring matcher. This way, when a given matcher fails, some of the subscriptions it previously owns can be reclaimed by the remaining neighbor matchers.

2) The client, upon detecting the failure of its current matcher, will re-initiate the join process and find the a new matcher to re-register its subscription interest. This is similar to a complete re-join, except that the spatial partitioning of the regions will have changed.

The first mechanism allows a fast re-join for clients of a failing matcher; the second mechanism ensures that recovery will eventually occur, in case the first mechanism does not work. Note that a matcher failure only has localized effects, as only its closest neighbors (i.e., the enclosing matchers) are affected and need to reclaim ownerships for the subscription records from the failed matcher. Intuitively, matcher failures are ultimately recoverable if the clients of a failed matcher attempt to re-join the system. However, from a quality of service perspective, it is still desirable if a matcher failure is almost unnoticeable to the clients.

IV. VSO Procedures

With the above rules, a Voronoi partitioning can be in constant adjustment of its region shape and size, such that the number of clients within a region will follow the matcher's capacity. Below, we describe the procedures of VSO in more details, but first summarize the terminology that will be used:

client a user node that performs publish/subscribe requests.
matcher a manager node of a Voronoi region that records subscriptions and performs pub/sub matching.
enclosing matchers matchers whose Voronoi regions surround a given matcher.
candidate matcher a node with sufficient capacity and reachability to qualify as a new matcher.
owner matcher a matcher whose region covers the center of a particular subscription.
acceptor matcher an existing matcher in the system that accepts a joining matcher.
gateway the first matcher of the system, and the keeper of a list of candidate matchers for matcher promotion.
replica subscription a subscription record replicated at another non-owner matcher whose region overlaps with the subscription area.

A. Initialization

1. The first matcher joins the system and acts as a well-known gateway for the system.

B. Matcher Join Procedure

1. A joining matcher sends a join request to the gateway.
2. The gateway determines whether the joining matcher is a candidate matcher and records it.

C. Matcher Leave Procedure

1. If there are no subscriptions managed by a matcher for an extended time, the matcher leaves the system by itself.
2. Enclosing matchers of the leaving matcher adjust their Voronoi regions using VON's Leave Procedure [13].
3. The leaving matcher repeats the Matcher Join Procedure to rejoin the pool of matcher candidates.

D. Subscribe (Client Join) Procedure

1. A joining client notifies an existing matcher of its desired subscription area. The existing matcher may be known from previous logins, or can simply be the gateway if no known matchers exist.
2. The receiving matcher forwards the subscription request greedily to find the owner matcher of the subscription.
3. The receiving matcher notifies the client of its owner matcher.
4. The client requests subscription from its owner matcher, which records it.
5. The owner matcher may forward the subscription to its neighbors to record as replica subscriptions, if the subscription area overlaps with neighboring regions (via VoroCast [16]).

E. Publish Procedure

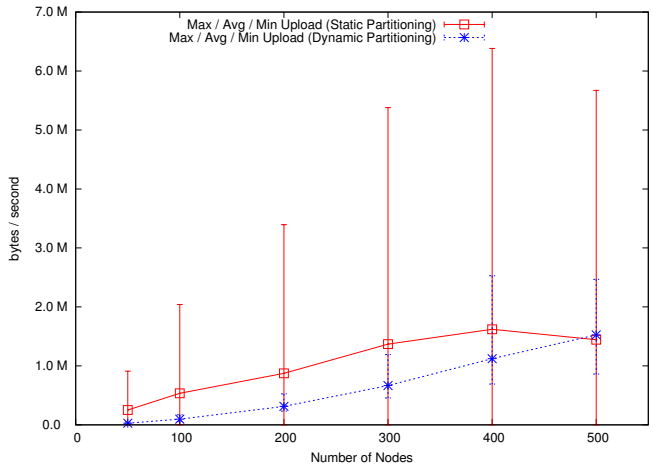
1. A client sends a publication to its owner matcher.
2. The receiving matcher checks if the publication matches one of its subscription records, if so, the owner matcher of the subscriber is notified. Note that the owner can be the receiving matcher itself and thus no actual transmission is made.
3. If the publication area overlaps with the regions of the enclosing matcher, it is forwarded to the enclosing matchers (using VoroCast [16] to avoid redundant forwarding).
4. Step 2 and 3 are repeated until no more matchers are covered by the publication area.
5. The owner matcher of the subscription notifies the subscribing client of the publication, while ensuring that only one notification is sent.

F. Load Balancing Procedure

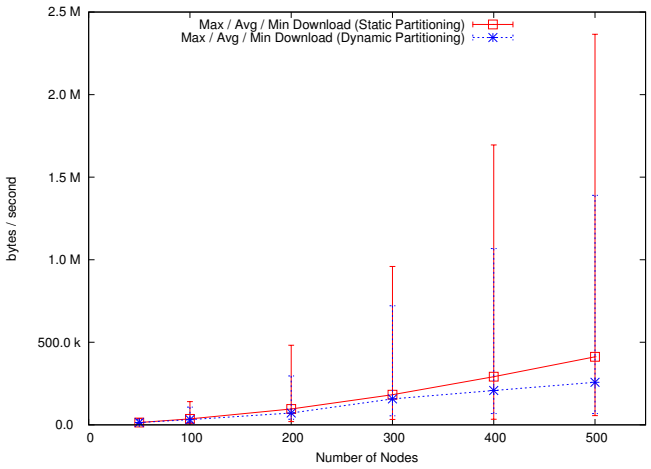
1. If a matcher is overloaded (based on the number of subscribers, or the level of activities), it sends a MOVE request to its enclosing matchers.
2. The enclosing matchers will move their sites closer to the overloaded matcher (see formula (1) in Section III-A).
3. If the overload situation does not improve after requests have been sent VSO_INSERTION_TRIGGER times, a MATCHER_INSERT request is sent to gateway.
4. The gateway selects a capable matcher from the candidate matcher list, and sends a PROMOTE request to the selected candidate. The request consists of the candidate's joining position, which is the load center (i.e., the center of all the subscription centers) of the overloaded matcher.
5. The selected candidate joins the matcher overlay via VON's Join Procedure [13] (i.e., asking gateway and be greedily forwarded to the closest acceptor matcher near its join location, where the acceptor matcher sends a list of initial matcher neighbors to the joining matcher.)
6. The joined matcher notifies the gateway of its successful join, to be removed from the matcher candidate list.

G. Ownership Transfer Procedure

1. Each matcher checks if the center of any recorded subscription is now outside of its region, if so, it sends a TRANSFER request to the respective matcher neighbor.
2. The neighboring matcher records the subscription and sends back a TRANSFER_ACK message to the previous owner matcher to confirm the transfer.
3. The previous owner matcher notifies the respective client of the change in ownership.
4. If the original owner matcher does not receive the acknowledgement within a timeout period, the ownership of the subscription is reclaimed.

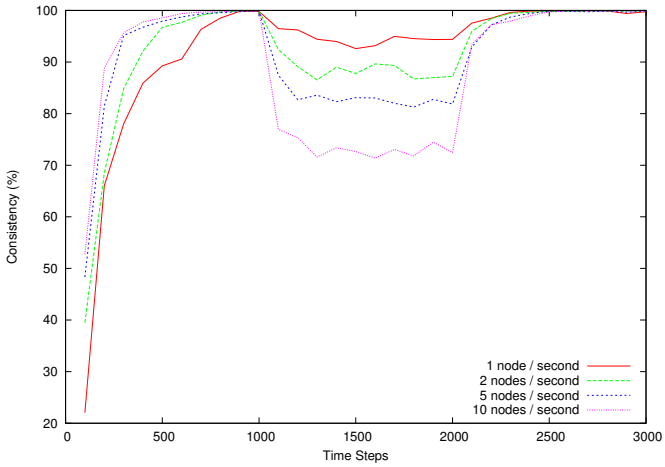


(a) Matcher Upload

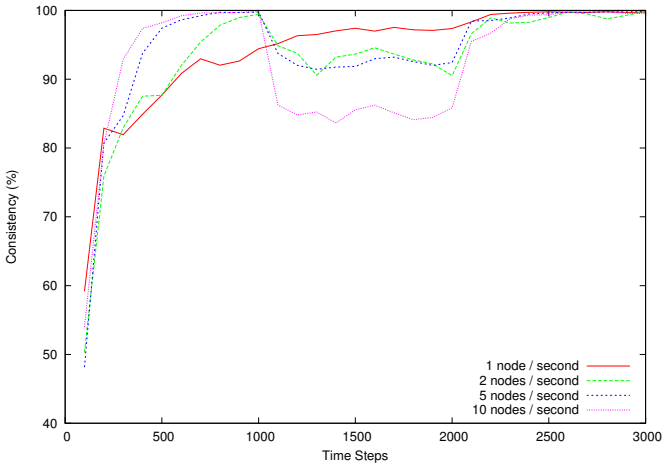


(b) Matcher Download

Figure 7. Static and Dynamic Partitioning Comparisons



(a) Stable Size 50



(b) Stable Size 100

Figure 8. Effect of Churn Rate on Discovery Consistency

V. Simulation Evaluation

We now present some preliminary evaluations of VSO based on simulations. For this initial evaluation, each client only performs area subscription and point publication of its position. We are mainly interested to answer these questions: 1) How does dynamic partitioning affect the work load balance? 2) How consistent and accurate is the pub/sub information, despite node churn? 3) How scalable can VSO be to accommodate high user density scenarios?

We perform the evaluation by extending the VAST library [13], which is an existing implementation of VON. All simulations are conducted in discrete time-steps of 100 ms each. For simplicity and to evaluate fully the bandwidth requirement, we assume no bandwidth limits and a constant latency of 100 ms between nodes (i.e., a message sent from node A to node B will be received and processed in the next time-step). The basic setup of our simulation is to mimic a Second Life region, with a 256 x 256 meters dimension, a user AOI radius of 64 meters [19], and a moving speed up to 10 meters / second. We multiply all dimensions by 3 to allow better visualization, and perform each simulation in the following steps:

1) A number of simulation nodes (i.e., clients) join the system at a certain join rate (i.e., joining nodes/second).
2) Each node moves with a clustering pattern within a 768x768 area at a speed of 3 units / time-step.
3) We record the maximum / average / minimum per second bandwidth usage at each node as they move.

The clustering movement is done by randomly placing 1.5 * $ln(n)$ hotspots (n is node size, so 100 nodes has 6 hotspots) where nodes would move towards the nearest hotspot with high probability. We allow each node to register as a potential matcher with the gateway (i.e., a P2P-like deployment), as pure client-specific bandwidth usage is often small. The overload limit for each matcher is set to 20 nodes (i.e., when subscriber size within a region exceeds 20, the matcher would start the load balancing procedure). To evaluate the correctness of the simulation, we adopt *discovery consistency* (or consistency for short) as the main metric. Discovery consistency is defined as the percentage of the nodes actually visible to a user from those that should be seen (e.g., if there are 10 nodes within a user's AOI, but only 9 are visible, discovery consistency is 9/10 = 90%) [13]. For all the simulations below, except the churn simulations, the average discovery consistency is always above 99%, indicating that the performance evaluation is based on the system first runs correctly.

Below we discuss our findings in the following three categories: 1) comparison of partitioning method; 2) performance under churn; and 3) scalability:

A. Dynamic Partitioning

We compare how dynamic partitioning affects matcher workload by first dividing the whole space into 9 regions and assigned them to 9 matchers. For static partitioning, the division is fully regular as 9 square regions. For dynamic partitioning, although the partitioning begins as square regions, region boundaries may adjust with time. To perform a comparable experiment, no new matchers are added to the system.

Fig. 7(a) shows the average, maximum, and minimum upload bandwidth of the matchers per second, as the total number of nodes increase from 50 to 500. We can clearly see that with dynamic partitioning, not only the average upload bandwidth is lower, the difference between the maximum and average upload is also greatly reduced. Workload thus is much more balanced. For matcher's download bandwidth, Fig. 7(b) shows similar results, where dynamic load balancing greatly reduces the workload's average and variance for each matcher. As matchers need to send updates to clients, their upload is greater than download in general.

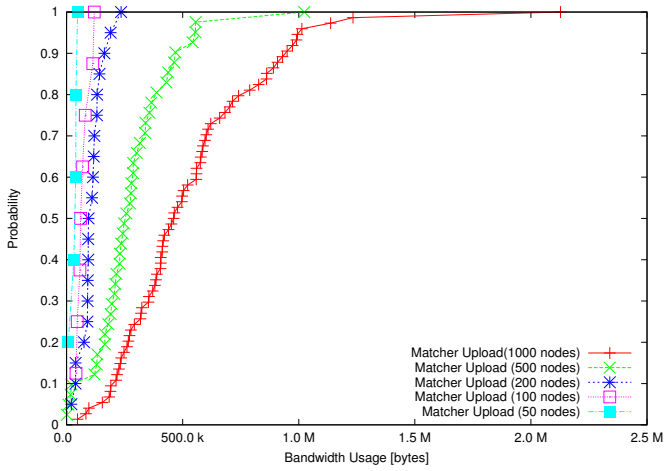
B. Effect of Churn

To evaluate how VSO performs under constant node joining and leaving (i.e., churn), we perform another set of simulations with three periods: joining, churning, and stabilizing. During *joining*, nodes enter the system at a constant defined rate and start moving; in *churning*, nodes would start to join and leave the system at a defined rate per second; during *stabilizing*, no node joins the system to allow stabilization.

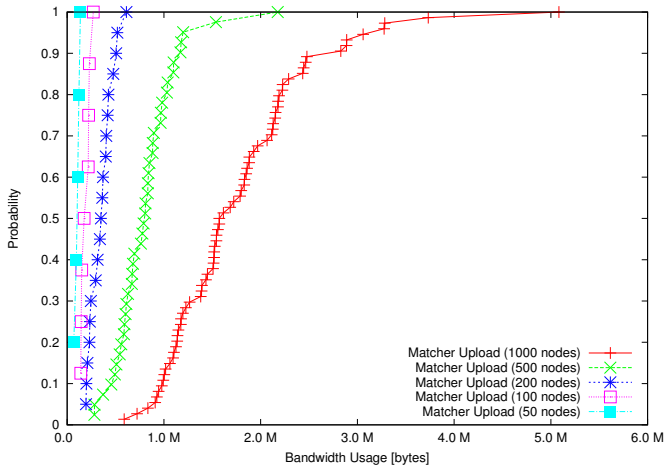
We set up a stable size of 50 and 100 nodes, and a churn rate of 1 node per second (i.e., there is one node joined and one node leave each second) to 10 nodes per second. The leaving nodes may be either a matcher or client, chosen at random. Fig. 8(a) shows the *discovery consistency* of the system as time goes, under 50 stable nodes. We can see that for the joining phase, the consistency gradually increases to over 99% until time-step 1000. As it takes time to join the system, nodes that should be visible may yet be learned, consistency thus may be low while nodes are joining. Once nodes are fully joined, starting from step 1000 to step 2000, churn begins and we see a drop in consistency to different degrees, the higher the churn rate the lower the consistency. After 2000 steps, we see that consistency restores eventually, mostly within 200 time-steps, or 20 seconds. The consistency drop can be as heavy as to 30% for a 10 nodes / second churn rate (i.e., when 20% of the nodes are constantly joining and leaving each second). Note that there is no hard limit on what consistency level must be in a given scenario, and it is rather application-dependent. Although we generally expect a discovery consistency of above 90% as being acceptable. In Fig. 8(b), a stable size of 100 nodes is used, and we see that churn's effects are less severe, likely as the number of churning nodes is a smaller percentage of the total nodes (i.e., 10% churn rate in the most serious case). We again see that the consistency drops when churn begins, and restores when there is no churn. The most serious drop becomes about 20%. The line for 1 node / second churn rate increases continuously as it takes some time for all nodes to fully join, but churning already begins before nodes are fully joined, thus the continuous increase. The important message from the two graphs is that: 1) consistency and churn rate have a negative correlation and 2) consistency restores back to normal eventually after some time when the system re-stabilizes.

C. Scalability

For scalability evaluation, we are mainly interested in the practical bandwidth requirement when a high density of users exists. For this purpose, we simulate between 50 to 1000 nodes totally, all with a high joining rate of 10 nodes / second, and plot the *cumulative distribution function* or CDF of bandwidth usage for both the matchers and clients. Note that having 1000 nodes in our simulated area is a severe density scenario (i.e., it is 10 times the density of the current Second Life region). As matchers in general need to send out updates on neighbors, their upload is greater than download and is often the main scalability bottleneck. Fig. 9(a) shows the CDF of the matcher's average upload. We see that with 500 nodes, 90% of the matchers has an average upload of 500 KB / sec, while for 1000 nodes, 90% of the matchers has an average upload of 1 M bytes / second. This means that a majority of matchers on average sends out less than 10 Mbps of data, which is within the range of

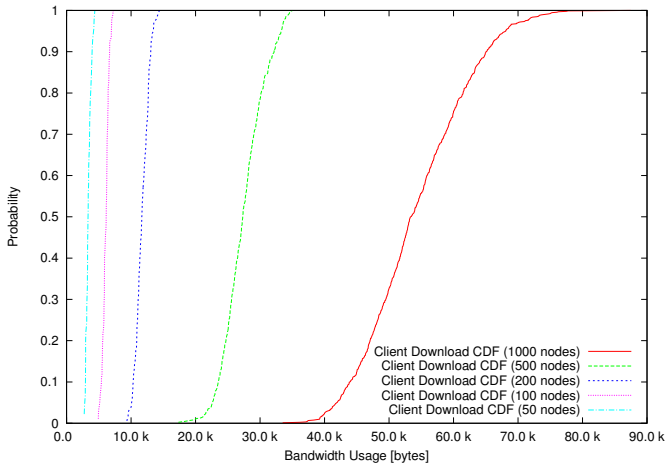


(a) Average Upload CDF

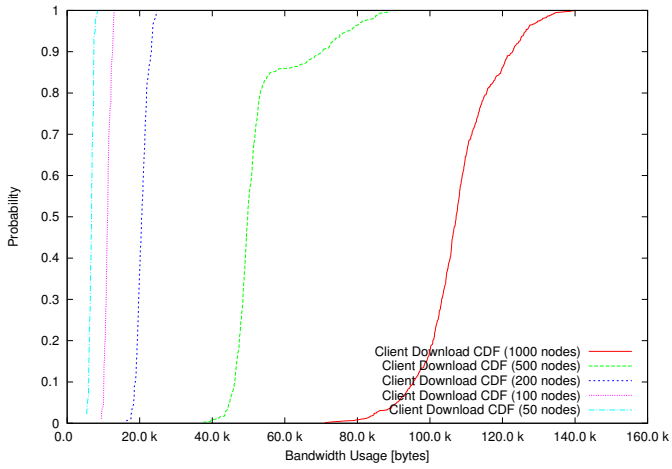


(b) Maximum Upload CDF

Figure 9. Matcher Per Second Upload CDF



(a) Average Download CDF



(b) Maximum Download CDF

Figure 10. Client Per Second Download CDF

home ADSL. However, to support actual systems, maximum uploads should be considered and are shown in Fig. 9(b). Here we see that the maximum upload is between two to three times the average upload. For a 500 node region, 90% of the matchers can be supported with an upload of 1.2 M bytes / second, while all can be supported by an upload of 2.2 M bytes / second (or 20 Mbps upload). For 1000 node region, 90% of matchers are supportable by 2.8 M bytes / second upload, and 100% are supportable with 5.0 M bytes / second upload. Our simulation results thus imply the following: 1) the required bandwidth resources for matchers is supportable with a combination of low and high bandwidth hosts, where the majority (90%) can be of relatively low upload capacity (i.e., ADSL-grade). For example, in the case for 500 nodes, ADSL with 10 Mbps upload can cover 90% of the matchers' upload needs; 2) In general, the matcher size is only 10% of the total number of nodes in the system (see Table I), so if ADSL-grade hosts can be the matchers, it is possible to support a whole region of relatively dense user population (e.g., 500 nodes) with client machines, where only 10% of them need to assume matcher responsibility. Given typical ADSL bandwidth distributions and user incentives [20], such is a likely scenario.

Fig. 10(a) and Fig. 10(b) show the CDF of the average and maximum download bandwidth for clients. we can see that a majority of clients (90%) uses less than 120 KB / sec of download (less than 1 Mbps), even for the 1000 node scenario. Clients in general thus are supportable with today's ADSL environment, even for dense user populations.

Table I
MATCHER RATIO

Node Size	Matcher Size	Matcher Ratio
50	5	10.00%
100	8	8.00%
200	20	10.00%
500	41	8.20%
1000	74	7.40%

VI. CONCLUSIONS

We present the design and evaluation of Voronoi Self-organizing Overlay (VSO), an adaptive mechanism to adjust workloads within Voronoi regions to support spatial publish subscribe (SPS) services. VSO utilizes two basic mechanisms for load balancing: boundary shifting and new region (i.e., matcher) insertion. Tolerance to matcher or client failure is also supported by making backup subscriptions and having client rejoin mechanisms during matcher failures.

We show from simulations that VSO reduces both the average and variability of the workloads for matchers, so that overload is less likely to occur, resource provisioning can also be more dynamic. Fault tolerance to churn up to 10% in a 50 node region is supported, and the overlay can recover from churn. Scalability analysis shows that even under a very dense scenario of 1000 nodes per region, 90% of the matchers only need an upload of 1 M bytes / sec (i.e., 10 Mbps upload) on average. While 10 Mbps is enough to support the maximum upload requirement for 500 nodes per region for 90% of the matchers. In other words, home ADSL connections plus a few server resources is sufficient to support fairly high-density interactions.

While the initial evaluation is promising, we would still like to evaluate VSO in more realistic environments and under full SPS operations (e.g., area publications). The dynamic partitioning and load balancing of VSO is general enough to be deployed in both cloud or P2P environments. How to best utilize both resources for scalability and performance is thus an interesting question. Another important aspect to investigate is whether under highly fluctuating workloads, the number of active matchers can indeed adjust accordingly to conserve resources.

REFERENCES

[1] S.-Y. Hu and K.-T. Chen, "Self-organizing spatial publish subscribe," in *Proc. ICAC (poster)*, 2011.

[2] S. Singhal and M. Zyda, *Networked Virtual Environments: Design and Implementation*. ACM Press, 1999.

[3] S. Douglas, E. Tanin, A. Harwood, and S. Karunasekera, "Enabling massively multi-player online gaming applications on a p2p architecture," in *Proc. ICIAD*, December 2005.

[4] A. R. Bharambe, S. Rao, and S. Seshan, "Mercury: A scalable publish-subscribe system for internet games," in *Proc. NetGames*, 2002, pp. 3–9.

[5] C. GauthierDickey, V. Lo, and D. Zappala, "Using n-trees for scalable event ordering in peer-to-peer games," in *Proc. NOSSDAV*, June 2005, pp. 87–92.

[6] B. Knutsson, H. Lu, W. Xu, and B. Hopkins, "Peer-to-peer support for massively multiplayer games," in *INFOCOM*, 2004, pp. 96–107.

[7] S.-Y. Hu, "Spatial publish subscribe," in *Proc. IEEE Virtual Reality (IEEE VR) workshop Massively Multiuser Virtual Environment (MMVE'09)*, 2009.

[8] K. Morse, L. Bic, and M. Dillencourt, "Interest management in large-scale virtual environments," *Presence*, vol. 9, no. 1, pp. 52–68, 2000.

[9] 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 Workshop NIME*, 2008, pp. 1134–1138.

[10] K. L. Morse and J. S. Steinman, "Data distribution management in the hla: Multidimensional regions and physically correct filtering," in *Proc. Spring Simulation Interoperability Workshop*, 1997.

[11] A. Bonotti, L. Ricci, and F. Baiardi, "A publish subscribe support for networked multiplayer games," in *Proc. IMSA*, 2007, pp. 236–241.

[12] E. Lety, T. Turletti, and F. Baccelli, "Score: A scalable communication protocol for large-scale virtual environments," *IEEE/ACM Trans. Networking*, vol. 12, no. 2, pp. 247–260, 2004.

[13] S.-Y. Hu, J.-F. Chen, and T.-H. Chen, "Von: A scalable peer-to-peer network for virtual environments," *IEEE Network*, vol. 20, no. 4, pp. 22–31, 2006.

[14] R. K. Balan, M. Ebling, P. Castro, and A. Misra, "Matrix: Adaptive middleware for distributed multiplayer games," *LNCS (Middleware 2005)*, vol. 3790, pp. 390–400, 2005.

[15] J. Liebeherr, M. Nahas, and W. Si, "Application-layer multicasting with delaunay triangulation overlays," *IEEE JSAC*, vol. 20, no. 8, pp. 1472–1488, 2002.

[16] J.-R. Jiang, Y.-L. Huang, and S.-Y. Hu, "Scalable aoi-cast for peer-to-peer networked virtual environments," in *ICDCS Workshops*, 2008.

[17] L. Genovali and L. Ricci, "Aoi-cast strategies for p2p massively multiplayer online games," in *Proc. IEEE CCNC*, 2009.

[18] M. Steiner and E. W. Biersack, "Shortcuts in a virtual world," in *Proc. CoNext*, 2006.

[19] H. Liang, M. Motani, and W. T. Ooi, "Textures in second life: Measurement and analysis," in *Proc. P2P-NVE*, 2008.

[20] A. Bharambe *et al.*, "Donnybrook: Enabling large-scale, high-speed, peer-to-peer games," in *Proc. SIGCOMM*, 2008.