Spatial Publish Subscribe

Shun-Yun Hu
Department of Computer Science and Information Engineering
National Central University, Taiwan, R.O.C.
syhu@csie.ncu.edu.tw

***Abstract*—Publish / subscribe is a well-known mechanism that allows entities interested in certain information (i.e., the *subscribers*) to receive relevant messages from some message generators (i.e., the *publishers*). We argue that for networked or distributed virtual environment (VE) applications, a *spatial publish / subscribe* (SPS), where each entity can receive messages generated within a specified space in the VE, is a fundamental mechanism underlying various VE applications. It is therefore of importance for the VE community to understand the main characteristics and limitations of SPS. This paper describes the basic mechanism of SPS, and how it can be used to support existing VE requirements such as overlay management, state management, and content management. Issues for implementing SPS are also discussed for potential developers to consider.**

I. INTRODUCTION

Massively Multiuser Virtual Environments (MMVEs) such as Massively Multiplayer Online Games (MMOGs) have become very popular in recent years. A fundamental characteristic for MMVEs is their scalability, where concurrent users in a single world instance may be between thousands (e.g., 2,500 users in an EverQuest world) and tens of thousands (e.g., 45,000 peak concurrent users in EVE Online and Second Life). Although the total number of online users for a MMOG may be over a million (e.g., China's World of Warcraft passed the 1 million peak user mark in 2008), it is realized by replicating many parallel world instances, each of which has a few thousand maximum number of users, who cannot migrate or communicate across worlds. To further increase MMVE scalability, researchers have since investigated peer-to-peer (P2P) architectures [1]. However, P2P designs face security and reliability issues, given the unstable and heterogeneous nature of user machines. Commercialization thus is only at the beginning[1], with visible success yet being seen.

Regardless of a client-server or P2P architecture, certain fundamental operations are required by any VE applications. We identify such operations as *spatial publish / subscribe* (SPS), and argue that most existing VE functions can be reduced to SPS operations. An implication of this assertion is that if SPS can be made highly scalable, a virtual world with *millions* of concurrent users may become possible.

In a distributed system, the basic publish / subscribe (or pub/sub) paradigm [2] is that users or *nodes* interested in specific messages (i.e., the *subscribers*) would first announce their interests via *interest expressions* (in the form of *subscription request* messages). Later, when messages are produced by message generators (i.e., the *publishers*), they would be matched against previously announced subscriptions, and delivered to the relevant subscribers. In a client-server architecture, the server can act as a message hub to easily support such function, where publications received by the server are checked and delivered to other clients who have expressed interests. Two main types of pub/sub models are channel-based and content-based [2]. *Channel-based* (also known as *subject-* or *topic-based* [3]) has been the most basic type of pub/sub, where users would subscribe to specific *channels* as interest expressions, and receive all messages published to the specific channels. *Content-based* pub/sub filters published messages according to the message content, and thus provides greater flexibility at the cost of more processing and resource usage.

We observe though, most operations in a VE require a different type of pub/sub than channel-based pub/sub. In its most basic form, a VE allows each user to assume a virtual representation called the *avatar*, who needs to be aware of other users and events within its visibility (called the *area of interest*, or AOI of the avatar). A classical approach thus divides the VE into regularly spaced grid or hexagon cells, each assigned a multicast channel (either IP- [4] or application-layer-based [1]). Users then simply subscribe to the cells that overlap with their AOI to receive relevant message updates. As users move, they will need to continuously subscribe to new cells and unsubscribe from old ones. However, the subscribed areas are often larger than the actual AOI, causing extra messages to be received. On the other hand, if the cells are made smaller to avoid extra messages, then more subscription maintenance is needed, as more cells are needed for contact, producing another overhead [2], [5].

A more intuitive method thus is for each user to specify a subscription just once in the form of a space that covers the AOI exactly (i.e., a *spatial subscription*). The subscribed space can move automatically along with user movements. Relevant messages published by a generator node (i.e., a *spatial publication*) would then be sent to the subscribers. As shown later, this mechanism is sufficient to support almost all existing VE functions. Overlay, state, and content management can also be built on top of such a primitive.

In the following sections, we first introduce some background in Section II on VE requirements and various variants of pub/sub for VEs. Section III describes the basic SPS model. We will give examples in Section IV on how SPS supports existing VE functions, and discuss implementation issues in Section V. The paper concludes in Section VI.

[1]http://www.badumna.com/

II. Background

We first briefly describe the basic requirements and mechanisms of a VE. VEs can be seen as state machines where many *game states* (i.e., attribute values that can be changed) stored inside *game objects* are updated based on *event* messages and *game logic* (i.e., rules of the game). Event messages can be generated by users (which we refer to as *actors* [6]) as they act (e.g., walk, pick up, or use certain items), or by internal game logic (e.g., a grenade has collided with a wall). The events are processed by a certain *arbitrator* that would update the game states by following the rules dictated by game logic. Note that to ease implementation in practice, the arbitrator may be implemented as processing separate event queues (instead of a master queue) for each game object in sequence [6]. Once game states are changed, *update* messages are then sent to each actor whose view is affected. For example, in client-server VEs (e.g., MMOGs [7] and first-person shooter, or FPS games [6]), each user machine is an actor, and a server is assigned as the arbitrator. Users generate events that are sent to the server for processing, while the server sends back updates to each user machines to be displayed (often just the relevant updates within the user's AOI). In fully connected peer-to-peer VEs (e.g., real-time strategy, or RTS games [8], or military simulators such as NPSNET [4]), each user machine serves as both the actor and the arbitrator. So each machine would receive *all* the event messages, process them, and reflect the changes locally, without being notified by a central authority.

To scalably support this *event - process - update* cycle, existing proposals can be categorized by how actor nodes learn of the updates within their AOI. The first set of proposals is the *spatial multicast*, where update messages are multicast to a group of receivers. The earliest approaches utilize multicast channels, where the VE is divided into some rectangular or hexagonal regions, each assigned a multicast channel. Each actor node can subscribe to the channels of the regions overlapped with its AOI, to receive any updates sent to the channel (from other actors or the arbitrator). This approach faces the inherent difficulty of determining the right region size: too large a region delivers excessive messages to each actor nodes, while too small a region necessitates many subscription requests, generating message overheads (e.g., NPSNET [4] and VELVET [9] use IP multicast, whereas more recent work such as SimMud [1] uses application-layer multicast).

To address the inadequacy of channel-based multicast, spatial multicast delivers messages to only a specified area in the VE (e.g., N-trees [10] sends message to a specified area, while Solipsis [11] and VON [12] receive messages from a certain area). Regardless the support is for publication or subscription, spatial multicast delivers messages to precise targets, avoiding the message overhead due to the the approximate nature of channels. However, spatial multicast is not flexible enough to allow varying degree of message receival: if a message is published to an area, all actors within the area will receive the message regardless of interest; likewise, if an actor subscribes to an area, it will also receive messages without considering the potentially different publication ranges (e.g., the sound of an explosion should be heard farther away than a chat).

A second category of update propagation has utilized *spatial query* as a fundamental primitive (e.g., OPeN [13], Colyseus [14], and HyperVerse's GP3 [15]). Spatial query allows a node to register certain game states at a location, then query a specified area for these states some time later. It provides some persistency of the game states that can be queried multiple times by existing nodes or new nodes to the system (i.e., addressing a *late joining* scenario). However, querying may take $O(log\ n)$ time (where n is the number of total nodes responsible for answering the query). Also, in between queries, newly registered updates may not be queried / discovered timely enough. Spatial query thus still has limitations in the total number of supportable nodes, and latencies that may not meet the real-time requirement of VE applications.

Few works related to VE have explicitly stated the use of publish / subscribe as an underlying mechanism. Of note are the communication architecture proposed by Fiedler et al. [16], the original Mercury [2], DiGAS [17], and HLA's DDM [18]. Fiedler et al. propose to use grid-based partitioning for the VE and channel-based pub/sub to receive events from other actors. So each actor would receive relevant events from the grids overlapped with its AOI. This has the basic drawbacks as other channel-based multicast methods. The original Mercury is basically a content-based pub/sub system, where subscribers can expressed interests in a flexible, SQL-like subscription language. Ranges in any given attribute field (e.g., between 50 to 100 in the x-coordinate of a *position* field) can be specified as the subscription. As such, filtering in Mercury is quite powerful and can extend to any content value that has strict ordering. However, to provide such flexibility, a lookup (i.e., a spatial query) with $O(log\ n)$ time has to be performed continuously, whose latencies may become unacceptably large when the node size increases. Our observation is that by focusing pub/sub on only the spatial domain, we may utilize certain optimization for better performance. DiGAS proposes a spanning-tree-based *broker network* to deliver and filter pub/sub messages. Subscriptions are broadcasted to *all* brokers so that once a publication is received, any broker can route the published message. Division of the VE into cells helps to make subscriptions more compact (only cell numbers need to be specified as subscriptions). However, a broadcast-based subscription may not scale for a larger node size. Here, we note that if each broker has clearly defined responsibilities, publications may then be delivered via some kind of targeted, directional routing [3], reducing much the control overhead to maintain the pub/sub mechanism. High Level Architecture (HLA) is a U.S. military and IEEE standard for simulation interoperability. Its Data Distribution Management (DDM) allows simulation nodes (called *federates*) to specify *subscription regions* over object attributes to receive updates sent through *update regions*. DDM may be the closest example of a spatial publish / subscribe. However, like Mercury, DDM's filtering is also very general and can be done over any attribute values, making its efficient implementation non-trivial.

III. A Basic Model for SPS

We argue that neither spatial query nor spatial multicast satisfy sufficiently the basic requirements of MMVE systems, where messages need to be delivered with minimal control overheads and latencies. The excessive message overheads of multicast, due to the inherent difficulty to choose the right region size, makes multicast inflexible to use [5]. The excessive latencies due to the nature of query, and in between successive queries, also makes spatial query inadequate for MMVE's real-time constrains [19]. From another perspective, query and multicast both represent needed aspects of a complete system: query indicates a subscription interest (i.e., receiving messages from all nodes within an area), while multicast indicates a publication interest (i.e., delivering messages to all nodes within an area). Both aspects can also be understood as an entity's awareness (or *focus*) and effect (or *nimbus*) about its surrounding [20]. Spatial publish / subscribe thus may be a more suitable and complete mechanism. In fact, spatial multicast and spatial query can be easily supported by a spatial publish / subscribe mechanism, as we will show later.

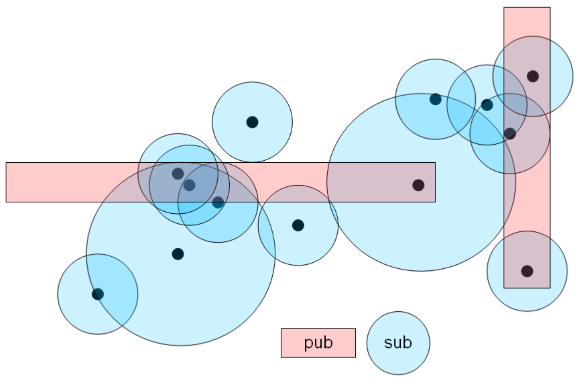

Fig. 1. Schematic for a 2D Spatial Publish Subscribe (SPS). In a military example, black dots may be soldiers, and publications may be weapon ranges, while subscriptions may be the views of soldiers or radars. Note that a pub/sub area can be of arbitrary size, shape, or direction.

We define a *spatial publish / subscribe* (SPS) as a mechanism where each *node* within a distributed system can specify a *publication space* and a *subscription space*. A node may send a message to its specified publication space, which will then be delivered to all nodes whose subscription spaces overlap with the publication space (Fig. 2). We further allow these pub/sub spaces be updated continuously with node movements (i.e., a new space overlaps mostly with a previous space). Publication and subscription requests are sent from a participating node to an *interest matcher* whose main responsibility is to record the requests and match a published message with potential subscribers interested in the message (i.e., performing *interest management* [21] between the publishers and subscribers). Note that the SPS concept is general enough to be applied to spaces of any dimension. However, for simplicity, we will mainly discuss 2D spaces for this paper, and may refer to a publication / subscription space equivalently as a *publication / subscription area*.

In a 2D SPS, there are thus four basic operations: *point publication*, *area publication*, *point subscription* and *area subscription*. A point publication is to publish a message at a given spot, receivable by all nodes whose subscription areas cover the spot; an area publication delivers a message to all nodes whose subscription areas or points overlap with the publication area; a point subscription allows a node to receive all messages whose publication areas cover the subscription point; and an area subscription allows a node to receive all messages delivered by publication areas or points that overlap with the subscription area. From another view, a point publication is an area publication with a zero size; likewise, a point subscription is an area subscription with zero size.

SPS can be seen as a specific form of *content-based* pub/sub [2], [3], where the filtering criteria is based only on the spatial aspect of the published messages, instead of other possible criteria (e.g., experience points and levels of the avatars). As such, we may divide and route the task of interest matching very efficiently according to spatial domains, which is not necessarily applicable when the whole message content is subject to filtering (as done by Mercury [2] or DiGAS [17]). In other words, instead of supporting a general content-based publish/subscribe, by focusing only on the spatial filtering of published messages, we can gain certain advantages in design simplicity and execution efficiency.

Unlike spatial query, a message receiver in SPS need not query continuously at different intervals to receive message updates, once a subscription is made. Unless the subscription area has changed, the message receiver need not re-send any requests. The delay between the occurrence of an update and the discovery of the update thus is reduced. Take user positions as an example, in a spatial query approach (e.g., Colyseus [14] or GP3 [15]), a moving actor node needs to register its new positions at a recorder every once in a while (e.g., a P2P overlay in Colyseus, or a server in GP3), and other users must continuously query for updated positions within their AOI neighborhood (i.e., performing *neighbor discovery* queries), which will either: 1) not allow the most timely position updates due to the query intervals or the time taken by the query itself, or 2) cause excessive query messages be generated if query intervals are too short apart. In an ideal SPS, after a subscription request is made, no further continuous queries are needed, message overhead thus is reduced. A published message will also reach a subscriber directly, so the delay between message generation and receival can be minimized. The only disadvantage of a pure SPS against spatial query is that SPS does not retain states, so if some already published messages are required again (e.g., for a late joined node), additional measure is needed. Luckily, we could use a *state manager* to keep copies of the published messages or states, and re-send those messages/states to any node interested in past publications, as will be shown later.

Unlike channel-based multicast, where nodes within an affected area will receive multicast messages regardless of interests, SPS allows a more flexible and fine-grained approach to message delivery. Often the receivers in a multicast would either receive too many messages beyond their AOI (if the region size is too large), or subscribe to many multicast channels and face frequent channel switching (if the region size is too small). Although spatial multicast provides a more fine-grained messaging than channel-based multicast, the same message receiver or sender cannot choose to have varying messaging ranges. For example, a foot solider likely has an AOI smaller than a ground radar, and thus might only realistically observe nearby entities. In a spatial multicast, anyone within the multicast range would receive the message regardless of interest, so a moving entity cannot be observed only by a distant radar but not the nearer soldier (who has a smaller AOI and should not see the entity, see Fig. 2). By specifying subscription areas, a node in SPS has full flexibility to decide the types and the amount of messages to receive.

On the other hand, spatial multicast and spatial query can be easily supported by SPS. Spatial multicast is actually a special case of SPS, where all nodes perform area publications and point subscriptions (or alternatively, point publications and area subscriptions). Although spatial query cannot be directly emulated with SPS, as spatial query is stateful (i.e., states can be registered once and queried multiple times later), with the support of a *state manager* that keeps the last published (i.e., most current) message or state, we can have the state manager subscribe to an area, and return the latest messages or states to any node that has requested to catch up.

Our definition for SPS leaves out the details for its implementation. A trivial implementation can be done with a simple client-server architecture, where the clients are the message generators / receivers (i.e., publishers and subscribers), and the server is the interest matcher. All pub/sub requests thus are sent from clients to the server, which keeps the requests (i.e., the interest expressions) and matches the received publications with potential subscribers. One may also utilize *bitmaps* to describe the spatial interest expressions for fast matching [22]. How to perform SPS more scalably with multiple interest matchers (in a server-cluster or P2P overlay) is more sophisticated and requires more considerations. For example, the subscription records have to be divided among several interest matchers, and publication requests need to be routed to the respective interest matchers efficiently (instead of being flooded to *all* interest matchers as in DiGAS [17]). The division or partitioning of the spatial domain can be done in many different ways (e.g., regular grids, hexagons, strips, triangles, quad-trees, or Voronoi diagrams, see [19] for a list of references), with tradeoffs that may be application-dependent. One also has to take note that updates in subscriptions may take time before affecting future publications (e.g., a publication may reach an already expired subscription, or miss a newly registered subscription). So how to perform subscription updates efficiently and correctly is another major challenge.

IV. SPS Applications

In this section we will show the usefulness of SPS in supporting certain basic functions for VE systems, including overlay, state, and content management.

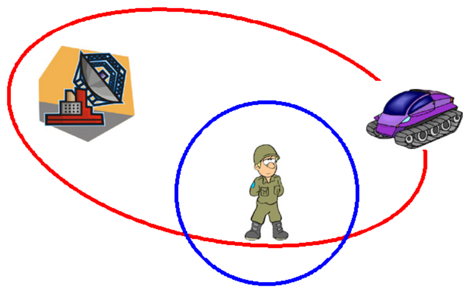

Fig. 2. Limitation of a spatial multicast. The tank's updates are sent to all nodes within a given area. If the eclipse and circle represent the radar's and the soldier's AOI, respectively, the incoming tank's update messages cannot be received only by the radar but not the soldier.

A. Overlay Management

Overlay management refers to the construction of P2P overlays in addressing *neighbor discovery* (also known as the *local awareness* property [11] or the *awareness problem* [23]) for each actor node in the system. The basic issue is: given the virtual position of an actor, find the positions of other actors within the given actor's AOI. Neighbor discovery has been the central issue since early P2P-based VE designs (e.g., Solipsis [11], VON [12], COVER [23], Colyseus [14], and the more recent GP3 [15]). One can provide neighbor discovery by utilizing a spatial query (as done by Colyseus or GP3) or by a spatial multicast (as done by Solipsis or VON). However, as mentioned earlier, spatial query introduces unwanted latencies, while spatial multicast may generate extra message overheads, and cannot support AOI of different interest ranges.

SPS supports neighbor discovery inherently in a fundamental and flexible way. Each actor node only needs to subscribe for messages within its AOI, and announces its position as a point publication whenever its position changes. As long as the subscribed area also changes with actor movements, each actor can learn about position updates as they occur. The discovery of neighboring actors can also be tailored individually with subscription areas of different sizes.

The only issue left is *late joining*, where a newly joined actor node needs to learn about existing neighboring nodes. With spatial query, this is addressed natively with the query; while with spatial multicast, late joining is solved by requiring each node to periodically send its current position. For SPS, late joining can be addressed by requiring each node to notify its own position to a new node whenever new nodes are discovered (via the existing nodes' area subscriptions). This way, neither a registration for the positions of all nodes (as needed by spatial query), nor a periodic re-send of positions (as needed by spatial multicast), would be required.

B. State Management

As mentioned earlier, VE systems can be understood as state machines that take *events* as input to modify the game states, and produce *updates* to notify nodes that are interested or affected by the changed states. This *event - process - update* cycle thus is a basic building block for VE systems.

To support state management, we can utilize two *layers* of SPS, one for events and another for updates. Each actor in the system (e.g., the client node) would perform an area subscription over its AOI in the update layer, and publish events as point publications in the event layer. The arbitrator (e.g., the server node, or a super-peer in a P2P architecture [19]) would perform pub/sub in the opposite manner. An arbitrator can perform an area subscription over its responsible region in the event layer to receive events; and it can publish updates as point publications in the update layer, at the locations of game objects whose states have changed. This way, the arbitrator would receive all the relevant event messages within its region of responsibility. After processing the events and updating the states, the arbitrator can then send updates back to the actors who might be interested. The arbitrator also need not worry about the different visibility ranges of the actor nodes, as nodes with different subscription areas (e.g., a small area for a foot soldier or a large area for a radar) would easily receive the messages proper to their scopes of interest.

This design naturally lends itself to a scalable state management where many arbitrators can jointly support a large-scale VE with many actors. Other design issues worth considering are cross-region consistency control, load balancing and fault tolerance for the arbitrator nodes [19]. To deal with late joining (i.e., a newly joined actor node learning about existing game states), the arbitrator can send existing states to a new node after detecting the new node's presence from the event layer.

C. Content Management

Content management is another recent focus in VE designs, and deals with game content that is becoming too large and dynamic. Most existing VE systems require users to pre-download or pre-install the content (e.g., 3D mesh models and textures, animations and voice files) on the local machines. However, it becomes inadequate when the content size grows from the range of a few mega or giga bytes, to terabytes (e.g., Second Life has over 34 terabytes of content in 2007). Some recent works (e.g., FLoD [24], LODDT [25], and HyperVerse [26]) thus propose to use P2P networks to offload the content delivery from server to clients. The basic idea is that as users in a VE often have overlapped visibility, when they are nearby to each others, the content required for download or rendering can be obtained from not just the server, but nearby peers as well. Current proposals basically consist of a *discovery* stage and an *exchange* stage. In the first stage, each peer attempts to find out which objects are to be downloaded, and which peers might have those objects. This can be done in either a distributed [24], [25] or centralized [26] manner. The second stage concerns with how to perform content exchange with the discovered peers, so that server resource usage can be reduced.

SPS may aid the discovery stage by requiring each peer to subscribe for its AOI and publish its current positions or content availability as point publications. This way, each peer can learn of nearby peers, and also their content availability from the SPS. Note that this is supportable with overlay management. To discover the objects to download, we can assign certain servers or super-peers as *object managers* that manage and maintain the object states within specific regions. Each object manager would perform an area subscription of its responsible region to learn about the presence of peers in its region. Whenever new peers arrive, the object manager can directly notify the peers of object presence. Note that the subscribed area needs to be slightly larger than the actual responsible area (in fact, at least one AOI-radius larger), in order to notify peers in neighboring regions whose AOI cover the region. As no queries are needed, new updates (e.g., the content availability of a certain peer) can be quickly delivered to peers who are interested. We could also use different layers of SPS for the discovery of different types of objects.

V. Implementation Issues

While SPS implementation is beyond our current scope, some partial SPS implementations already exist as good references (e.g., Chen et al. [27] propose a SPS that supports point publications and area subscriptions for intelligent location-based services). Here we briefly discuss two major issues: 1) how to partition the VE to assign interest matchers, so that the pub/sub messages can be routed efficiently, and 2) how often should subscriptions be performed and maintained, so that subscription message overheads and delays are minimized.

As mentioned earlier, existing partitioning methods can be utilized for the first issue (e.g., fixed partitioning such as grids and hexagons, or dynamic partitioning such as strips and Voronoi diagrams [19]). If neighboring interest matchers are mutually aware of each other, pub/sub requests can be sent to any interest matchers and be forwarded to the responsible one via *greedy forward* (i.e., always forward the message to the neighbor closest to a target location). For the second issue, a subscription should be updated whenever the subscriber has moved or changed its subscription. However, to avoid excessive subscriptions, they can be sent to interest matchers only periodically with a slightly increased subscription area [26]. In some cases, predictive subscriptions may also reduce the potential latency caused by routing the subscriptions.

VI. Conclusion

In this paper, we have identified *spatial publish / subscribe* (SPS) as a useful primitive to VE applications. In the most simple terms, SPS provides a node to subscribe to a specific area within a VE to receive messages, and allows a node to publish a message to an area, receivable only by subscribers whose previous subscriptions match (i.e., overlap) with the published area. We have given examples on how SPS can support overlay, state, and content management, and demonstrated how existing systems can be reduced to or modeled by SPS.

We believe that SPS is a fundamental construct underlying existing VEs. As such, understanding its applications and limitations is of importance to MMVE researchers and developers. SPS can be implemented easily with a single server, but creating a scalable SPS with multiple interest matchers is still an open problem, and may hold the key to build VE systems with millions or more concurrent users.

Acknowledgments

This work was supported by National Science Council, Taiwan, R.O.C. under grant 95-2221-E-008-048-MY3. I would like to thank Chuan Wu, Eliya Buyukkaya for helpful discussions, the anonymous reviewers for useful comments, and Prof. Shing-Tsaan Huang, Prof. Jehn-Ruey Jiang for their supports.

References

[1] B. Knutsson *et al.*, “Peer-to-peer support for massively multiplayer games,” in *INFOCOM*, 2004, pp. 96–107.

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

[3] A. Dattaz, M. Gradinariu, M. Raynal, and G. Simon, “Anonymous publish/subscribe in p2p networks,” in *Proc. IPDPS*, 2003.

[4] M. R. Macedonia *et al.*, “Exploiting reality with multicast groups,” *IEEE Computer Graphics and Applications*, vol. 15, no. 5, pp. 38–45, 1995.

[5] E. Lety *et al.*, “Score: A scalable communication protocol for large-scale virtual environments,” *IEEE TON*, vol. 12, no. 2, pp. 247–260, 2004.

[6] T. Sweeney, “Unreal networking architecture,” http://unreal.epicgames.com/network.htm, 1999.

[7] P. Rosedale and C. Ondrejka, “Enabling player-created online worlds with grid computing and streaming,” Gamasutra Resource Guide, 2003.

[8] P. Bettner and M. Terrano, “1500 archers on a 28.8: Network programming in age of empires and beyond,” Proc. GDC, 2001.

[9] J. C. Oliveira and N. D. Georganas, “Velvet: An adaptive hybrid architecture for very large virtual environments,” *Presence*, vol. 12, no. 6, pp. 555–580, 2003.

[10] C. GauthierDickey *et al.*, “Using n-trees for scalable event ordering in peer-to-peer games,” in *Proc. NOSSDAV*, June 2005, pp. 87–92.

[11] J. Keller and G. Simon, “Solipsis: A massively multi-participant virtual world,” in *PDPTA*, 2003.

[12] 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, 2006.

[13] S. Douglas *et al.*, “Enabling massively multi-player online gaming applications on a p2p architecture,” in *Proc. ICIAD*, December 2005.

[14] A. Bharambe *et al.*, “Colyseus: A distributed architecture for multiplayer games,” in *NSDI*, 2006.

[15] M. Esch, J. Botev, H. Schloss, and I. Scholtes, “Gp3 - a distributed grid-based spatial index infrastructure for massive multiuser virtual environments,” in *Proc. P2P-NVE*, 2008.

[16] S. Fiedler, M. Wallner, and M. Weber, “A communication architecture for massive multiplayer games,” in *Proc. NetGames*, 2002, pp. 14–22.

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

[18] 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.

[19] S.-Y. Hu, S.-C. Chang, and J.-R. Jiang, “Voronoi state management for peer-to-peer massively multiplayer online games,” in *Proc. NIME*, 2008.

[20] S. Benford and L. Fahln, “A spatial model of interaction in large virtual environments,” in *Proc. ECSCW*, 1993, pp. 109 – 124.

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

[22] S. Lynch, “A spatial publish/subscribe model for virtual worlds,” http://seanlynch.livejournal.com/13798.html, 2009.

[23] P. Morillo *et al.*, “Providing full awareness to distributed virtual environments based on peer-to-peer architectures,” *LNCS*, vol. 4035, 2006.

[24] S.-Y. Hu *et al.*, “Flod: A framework for peer-to-peer 3d streaming,” in *Proc. INFOCOM*, 2008.

[25] J. Royan *et al.*, “Network-based visualization of 3d landscapes and city models,” *IEEE CG&A*, vol. 27, no. 6, pp. 70–79, 2007.

[26] J. Botev *et al.*, “The hyperverse - concepts for a federated and torrent-based ”3d web”,” *IJAMC*, vol. 2, no. 4, pp. 331–350, 2008.

[27] X. Chen, Y. Chen, and F. Rao, “An efficient spatial publish/subscribe system for intelligent location-based services,” in *Proc. 2nd international workshop on Distributed event-based systems (DEBS)*, 2003, pp. 1–6.