ImonCloud: Easing Development and Deployment for Scalable Networked Games

Shun-Yun Hu
IIS, Academia Sinica, Taiwan, R.O.C.
syhu@iis.sinica.edu.tw

Matthew Lien
Imonology Inc., Taiwan, R.O.C.
BlueT@imonology.com

***Abstract*—The number of small, independent game developers has grown recently due to the widespread of game creation tools such as Flash and Unity3D. However, adding multi-user support is still uncommon due to the amount of server and network knowhow required. Small developers thus are confined to making simple single-player games. This paper presents ImonCloud, a scalable system designed to easily add multi-user and spatial interactions for otherwise single-player games. Developers only need to focus on writing the game logic in JavaScript, the game's deployment and scaling is then done transparently. Clients connect to servers closest to them, while game logic is executed at nearby servers to reduce latency. Server redundancy is also provided to ensure the operation's reliability. Built-in support based on Spatial Publish/Subscribe (SPS) also allows developers to perform interest management with ease. ImonCloud thus lowers the barrier of developing multi-user, spatial games that it becomes possible for small teams and even individuals to make scalable and globally deployable games.**

I. Introduction

There is a long history of how networked games are developed. Although originally single-player, when LANs and dial-up Bulletin Board Systems (BBSs) become available, simple client-server games allow multiple players to engage in Multi-user Dungeons (MUDs), where players could roam and role-play in connected rooms by typing text commands and reading descriptions.

When networked devices (e.g., modems and network cards) became common in early 90s, first-person shooter (FPS) appeared as a popular genre where tens of players can connect to a central host to engage in fast-paced actions [1]. All users send *events* to the server, which processes the events to modify the game states. The server then sends back *updates* to each player to display locally. Around the same time, real-time strategy (RTS) allows multiple players to engage in fully-synchronized games with lots of interacting units. As RTS often supports thousands of object units, sending state updates directly is not feasible for the typically limited bandwidth, so RTS games only exchange player events (i.e., commands) among all hosts, and each host performs its simulation locally (i.e., each host fully simulates the game). As Internet became widely adopted, both FPS and RTS also moved from LAN to more players across WAN.

In the mid-90s, Massively Multiplayer Online Game (MMOG) genre allows thousands of players engage in the same epic story. They provided deep social interactions that allow players to not only role-play in carefully crafted stories, but also engage in diverse types of social behaviors. MMOGs became the dominant online game genre in mid-2000 with Blizzards release of World of Warcraft (WoW), which had a peak of 11 million active subscribers.

Since 2008, casual web games utilizing social networks to allow game-play with real-world friends emerged as another form of dominant games. The rise of social games match well with the more connected, yet fragmented time nature of Internet users, while bringing more people to play games that were not traditional game-players.

From MUD, FPS, RTS, MMOG, to social games, regardless the games nature or audience, developing smooth interaction experiences has always been non-trivial to developers. Only the most technically advanced or resourceful developers were able to provide great multiuser experiences. Among the challenges are [2]:

Consistency: at the most basic level, networked games require players to see a shared view of the world, and experience interactions with more or less the same sequence and types. This implies that game states should be processed in the right order, and delivered or presented to players in the same sequence.

Interactivity: engaging games require players to see each others response quickly, though latency requirements differ among game genres. Actions games may need to deliver packets in less than 150ms, while MMOGs can allow latencies up to seconds. Ensuring latency bounds across geographies is especially challenging for globally deployed games.

Scalability: The usage and demand for todays Internet games differ greatly during the games lifetime, days in a week, and even hours in a day. Being able to support concurrent players at various scales thus is important for game operators.

Reliability: Commercially deployed games typically need to be available 24 by 7. Although typically there are scheduled downtime for servers to fix bugs or apply new content, unexpected downtime due to bugs (i.e., logic errors) or overloading are unacceptable to players and may ruin the games reputation and loyalty.

Persistency: While early FPS or RTS are session-based (i.e., no player records exist after the session ends), keeping player or world states intact throughout different sessions was the default design for MMOG and social games. Storing such records in a persistent database thus has already become standard practice in the industry.

Security: Game-play experiences should execute according to certain designed rules, though some players will always attempt to modify game rules or network packets for unfair advantages. Anti-cheating measures ensure that such attempts do not succeed. Another aspect is the protection of player account or personal information, though the latter is more generic to all information systems. Most game systems nowadays adopt a basic client-server model where clients can only send in user commands (i.e., events) but not execute any key game logic, to protect state updates as being authentic.

Developing or transforming a single-player experience into multi-player thus is not trivial, and may involve the following:

- Define game states and game logic to fulfil design requirements.
- Define network packets and protocol to convey events and state updates.
- Design security measures to minimize the impact of packet modifications.
- Find the right deployment configuration to minimize latency for the gamers.
- Partition the server workloads to scale the concurrent user size.
- Develop effective procedures to identify and test transmission and logic bugs.

These are not easy tasks for small developers to tackle, we thus seek to design an architecture and methodology that may ease these difficulties. This paper presents the design and architecture of the ImonCloud system, which is a commercial cloud platform developed by Imonology Inc., with technology transfer from Academia Sinica, Taiwan. The main contribution of this paper is the design of a practical system that integrates a number of research concepts to bridge the gap between state of the art research works in scalable networked games and actual industrial adoption of such research.

A. Generic Game Loop

We first explain a generic structure for multi-player games (see Fig.1). Networked games typically consist of multiple players (also known as *actors*) that follow an iterative processing loop executed continuously to feed in *events* (i.e., player actions) to the server (i.e., an arbitrator of state changes), processed by the *game logic* (i.e., rules on what events trigger what responses), to modify *game states* (i.e., variables storing the current internal states), and send back state *updates* (i.e., modification instructions to game states) to players to update their local views of the game. Note that typically a player sees only a local/limited view of the global states (such as FPS / MMOG), though for games such as RTS, all players have the same global view of the game states. So this event - processing - update cycle defines a typical game loop.

As a game developer, the three main tasks to deal with are: 1) identify the game states and how will they be stored (as data structures); 2) define how game states are modified according to game logic, and 3) how player actions can be encoded as events, while changes to the local player states can be encoded as updates.

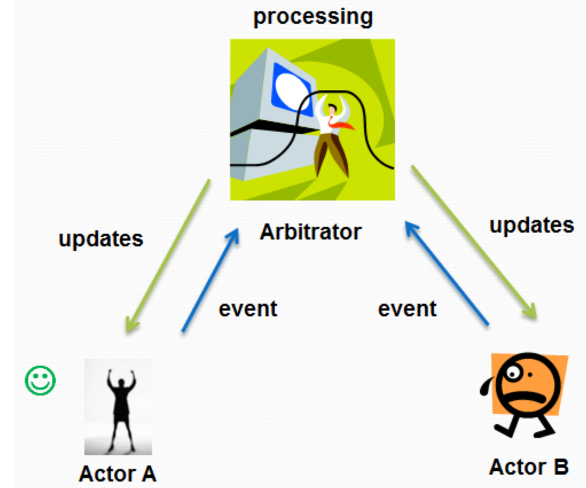

Fig. 1. event model in a generic game

B. Conversion Example

To demonstrate how a single player game may transform into a multiplayer game, let us use a simple tic-tac-toe game as an example. A typical gaming procedure involves two players in a tic-ta-toe game, each taking turns to send in their decisions to place the next move. A single-player game will require an AI module to play against the human player. So the game flow basically involves the following:

1) the user makes a move, captured as an event
2) the event is evaluated for correctness and state updates
3) the AI makes a move, encoded as an event
4) the event is evaluated for correctness and state updates
5) results of the evaluation is displayed
6) repeat the process or end the game

In order to make it into a multiplayer game, the above procedure may need to change into:

1) userA makes a move, captured as an event
2) userA's event is sent to the server
3) server evaluates the event for correctness and state updates
4) results of the evaluation is sent back to userA and userB
5) results of the evaluation is displayed at both users
6) userB makes a move, captured as an event
7) userB's event is sent to the server
8) server evaluates the event for correctness and state updates
9) results of the evaluation is sent back to userA and userB
10) results of the evaluation is displayed at both users
11) repeat the process or end the game

We can see that additional communications of events and updates are needed. If the game logic that processes user events can take events as inputs and produces updates as output, then the same single-player logic may also be used at the server. However, if not, then some additional works are required to modify the game logic to accommodate this model for multi-player interactions.

C. Practical Issues

Commercial games are often provided as a service with continuous operations, as such, gamers expect the system to be operational without unexpected downtimes. The system should also be secure, fair, and responsive. Such quality of service (QoS) guarantee is expected regardless of the current system load. As such, maintaining the same QoS despite of sudden user increase is the main scaling issue for online games. [3]

Latencies experienced by game players can be attributed to two sources: network latency (i.e., the transmission of packets over Internet) and system latency (i.e., the processing of the packets by applying game logic calculations). The former can be influenced by many factors typically not controlled by the game operator, especially if the game is deployed at a single site; while the latter may be minimized with good architectural design and engineering, to ensure that event processing is efficient, and the handling server does not become overloaded. Ensuring that the servers do not overload (whether by limiting user login, or performing load migration), thus are necessary to ensure the expected QoS.

II. ImonCloud Architecture

ImonCloud is designed to reduce the efforts to develop interactive applications by three main approaches: 1) provide a standardized way to write and execute game logic, such that the game logic can be deployed and scaled automatically and transparently without requiring networking knowhow or code changes; 2) provide automatic load balancing and connection by proximity so that users connect to the closest entry servers to join the system, operators thus do not have to worry about where and how to deploy optimally (i.e., code execution will migrate to where most users are located), and 3) provide spatial publish / subscribe (SPS) as a primitive API so that developers need not handle the difficult interest management. In short, ImonCloud provides developers with a unified development and deployment environment so that they can focus on game logic design and testing, while reducing their engineering efforts on development, deployment, and operations.

We leverage the rich body of existing research on scalable networked virtual environments (NVEs), especially the works based on peer-to-peer (P2P) techniques, as they provide various conceptual and theoretical ideas on how such a development platform can be built. Specifically we have leveraged three main ideas:

Object (entity)-based Partitioning For games with maps (e.g., MMOG, RTS, FPS), the map is often spatially partitioned by rectangles or hexagons to different regions, where each region is assigned to a particular server to distribute loads. This has the benefit of congregating all relevant states at the same host for computation. However, if the server fails, all region data managed will be lost. Alternative partitioning approaches have since been proposed [4], where the load is divided by entities in the system and assigned to arbitrary servers. As long as game states, game logic, and user events are available at the same processing node, user events can be processed correctly and game states updated accordingly. This design has the benefit that when a particular server fails, subsequent event handling can be simply re-routed to any remaining servers, without facing service interruption or game state loss.

Spatial Publish Subscribe Publish / subscribe (pub/sub) is a messaging mechanism where only subscribers of a given interest will receive messages from publishers that match the interests. This provides the benefit that subscribers or publishers need not know about each other [5]. The most basic form of pub/sub is channel-based, where publishers deliver message to a particular channel subscribed by interested parties. More advanced pub/sub may involve filtering based on more specific relations (i.e., content-based pub/sub). We utilize a specific content-based filtering called spatial publish/subscribe (SPS) [6], where publishers and subscribers specify their interests as 2D areas, so that a given message is delivered only if the publication area overlaps with the subscription area. An area can be rectangular or circular, and can also be of zero-radius (a point publication/subscription).

Voronoi Self-organizing Overlay (VSO) To provide SPS services, we utilize Voronoi Self-organizing Overlay (VSO) [6], which is a technique to divide a space into various regions based on Voronoi partitioning, such that the region size can be automatically adjusted according to the region's loading. The load is defined as the number of connected users to a given Voronoi region. Voronoi partitioning has the benefit that the least number of regions need to exist for a given load distribution. Also, as the partition hierarchy is flat (as compared to alternative partitioning such as trees), there is better fault tolerance if a given node fails to manage the partition.

A. SCALEM Component Model

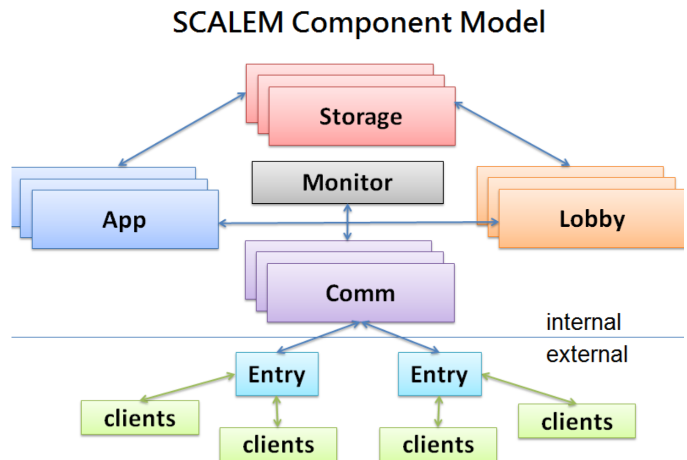

Fig. 2. the SCALEM component model for ImonCloud

We now describe the main components in ImonCloud, which can be collectively abbreviated with the SCALEM acronym, taking the first alphabet of each main component in the ImonCloud system:

Storage Important game states should be written to a database (DB) and stored permanently. We choose the NoSQL MongoDB for its flexibility and support of cluster configuration. We assume that the DB is scalable by its own design and can be logically seen as a single server. Note however that ImonCloud's scalability does not rest on the DB, but rather on the architecture of the logic execution environment.

Communication A unique and core design of ImonCloud is its adoption of a Communication Layer for all event/update traffic between clients and servers, and also among the servers. The layer provides both channel-based pub/sub and spatial pub/sub (SPS). By leveraging SPS for client-server communications, we effectively shield the client from knowing which server is processing its events, so that architecturally it becomes possible for different servers to handle events from the same client at different times (i.e., as in Object-based partitioning approaches). Likewise, the server has no knowledge of where the client is located or how to transmit updates to the client, simplifying the workload of the logic server.

App The main game logic is executed at the App Server, and it is assumed that each event can be correctly processed by the game logic, as long as the states and logic relevant to the event are available. So for a given incoming event, if an App Server can access and modifies all the relevant game states, then we can assign event processing to different App Servers, without impacting the results of logic processing, effectively balancing the loads among servers. This is similar to how web servers scale and handle increasing traffics. By moving game logic processing from a simple client-server model to more of a web model, App Servers are designed to scale to process more events as users or workload increases. This approach has traditionally been difficult to replicate for online games (especially games based on maps), because there is no straightforward methods to partition game workload and assign them evenly. ImonCloud attempts to achieve this even and balanced workload assignment via the Communication Layer.

Lobby The Lobby Server is the similar to App Server in that it executes game logic, with the difference that it also maintains certain game states of global (i.e., system-wide) relevance. For example, the full list of currently online users, and aspects related to accounts or payment.

Entry Entry servers are proxies to which the game clients connect and maintain connections. When a user client wishes to join the system, it first performs a short check to find the Entry Server with the lowest latency. It then establishes a connection to that Entry Server for the duration of the game session. All events and updates to/from the servers are communicated via this Entry Server, which may in turn communicate with the Communication Layer (e.g., to perform channel or spatial pub/sub).

Monitor Monitor servers are the ones that keep an eye on the liveness of other servers, and will re-start them if necessary. ImonCloud is designed such that the start/stop of any single server will not impact its ongoing operation.

B. Logic Execution Environment

We now describe how developers will build a networked, multi-user game using ImonCloud, and what tools and API are available for the development. Events are app-specific and may be defined by developers in the form of a JSON message, for example:

```
{
    E: 'MOVE_EVENT',
    P: {pos: {x: 12, y: 13, z: 0},
        speed: 16}
}
```

Fig. 3. a movement event data encoded in JSON format

Where E indicates the event name; and P indicates the parameters sent. The parameter is fully customizable by the developer. Events are received and processed by a currently live App Server. However, developers may also leverage SPS functions to limit how the event may be delivered or processed. For example, after receiving a login event, the relevant game logic may do the following:

```
Handler.LOGIN_REQUEST = function (event) {

    // check if the login is correct
    // "event.conn" is the connection object
    // "event.data" stores the JSON-decoded event parameters
    if (checkLogin(event.data.account, event.data.password) === true) {
        IC.SPS.subscribe(event.conn, event.data.pos, event.data.radius);
        event.done('LOGIN_RESPONSE', {result: true});
    }
    // if login fail, return a negative response
    else {
        event.done('LOGIN_RESPONSE', {result: false});
    }
}
```

Fig. 4. Code example for how a login request is handled

The above code allows this particular client to define and limit its interaction scope with an AOI, and will from that point on, only receive updates within its subscribed area.

C. Spatial Interest Management

For spatial games, each user is often interested in a particular Area of Interest (AOI) that centers on the user and moves with the user. ImonCloud provides five basic SPS primitives to perform spatial operations:

A user can subscribe a given area with a specific center point and radius, if the radius is 0 then it is a point subscription. Otherwise it can be an area subscription. Similarly, users can also publish to an area with 0 or non-0 radius (i.e., point publication and area publication). Both publish and subscribe operations are performed at a given layer, which provides logical separation of pub/sub interests within the same coordinate systems. That is, pub/sub requests will not be matched if they are on different layers. Publish/subscribe requests do not specify receivers, so a sender generally does not know who will receive a particular message. However, in situations when a sender may want to send messages to

```
// subscribe for an area at a given layer
subscribe(layer, area, onSubscribe)

// publish a message to an area at a given layer
publish(layer, area, msg)

// move a subscription to a new position
move(subID, area)

// send a message to specified user(s)
send(id, msg)

// get a list of subscribers within an area
list(area, onUpdate)
```

Fig. 5. interface definition for spatial operation methods

a particular user (for example, a one-to-one chat), the sender may use list to first get a list of neighboring users within a given area, then the user can use send to send a message to a particular user based on its unique id.

D. Load Migration

There are two mechanisms for load migration in ImonCloud, one is the partitioning of spatial workload using Voronoi partitioning, the other is the specific migration and setup of a new logic execution environment (i.e., App Server) when existing ones are overloaded. Of particular note is that our load migration is adjusted automatically based on actual workloads, instead of manual control. We utilize VSO [6] for the load migration, where the SPS functions are realized by utilizing a VSO. We describe the two load migration mechanisms in details below:

Spatial Messaging Spatial messages are processed by the Communication Layer using a SPS interface as previously described. The whole virtual world map is first divided into various regions, each managed by a *matcher* node. Matchers are basically servers that store and update the pub/sub info collectively and form a P2P network among themselves [6]. Matchers thus divide the entire map using Voronoi partitioning, such that each region has a center point of which the managing matcher may move. By communicating with neighboring matchers constantly about its own workload, a matcher can move toward or away from the center points of its neighbors, effectively modifying the shape and size of the region it manages. This way the loading (i.e., density of user subscriptions) can be controlled within a preferred limit in each region, achieving balanced load among the matchers. The collection of the matcher servers constitute the Communication Layer.

Note that because each matcher can set its own workload limit based on its resource availability such as CPU and RAM, the final managed region size may differ in sizes.

As all pub/sub requests are routed by the matchers, balancing the loads among matchers effectively achieve load balance for interest management as well. Matchers make P2P connections with their neighbors only on a need-basis, such that only if users within its managed regions have subscription interests in the areas managed by other matchers, will the matcher make a connection to it. As long as the user subscription area has a fixed upper size and the user density is being controlled, the number of neighboring matchers to connect will be bounded as well.

Code Execution Besides load migration for messaging (which is IO-bound), another factor to host loading is CPU usage. Actual CPU usage on each logic execution node (i.e., the App Server) has two components to it: number of users being managed and how complex their scripted behaviors are. While the first aspect may be controlled with the spatial messaging load migration, the second aspect is something controllable only by the developer.

However, ImonCloud can still migrate code execution as closely as possible to where most of the affected users are located, so that the response time of logic execution can be less impacted by how far the users and the App Servers are located, cutting the communication time.

The approach we use is to start the App Server first at a more or less random physical location, but monitors where most requests are coming from (requests may come from either an Entry Server directly, or passed by a node in the Communication Layer). The App Server would then monitor constantly how far away it is from the center of logic execution. That is, whether it is at a location close to most users sending requests to its logic.

If it is determined that there may be another App Server more suitable to execute the logic, it may then request that particular App Server to start executing the game logic and taking new requests from the users, while shutting itself down when the number of requests it processed is below a certain threshold. As all game logic are script based, they can be easily send between App Serves and executed on-the fly. New App Servers can join the logic execution by simply running up and joining at a virtual location with the rest of the App Servers. The above spatial messaging migration algorithm would then shift loads among the existing App Servers.

E. Deployment Setup

To allow Entry Servers provide the shortest possible latencies when user clients are interacting, actual Entry Servers should be deployed across different geographies. The assumption here is that by connecting to a physically close (i.e., with short latency) Entry Server, a client will have shorter round-trip latency with a Lobby/App Server at another geography, than if the client connects to the Lobby/App Server directly. This is because often times client-server connections can route through various paths of different qualities, whereas link qualities between servers are often more stable [7].

When a user connects to ImonCloud, there is a client-side network module that first contacts a few (at least three) Entry Servers from its cache, then a decision is made to make a permanent connection to one of the Entry Servers for this gaming session. Once a connection is established with the

Entry Server, all subsequent communications with ImonCloud are done over this connection.

The Entry Server then connects with the actual Lobby or App Servers to perform client-server communication (acting on the user clients behalf), though it can also make pub/sub requests to the Communication Layer, in case spatial operations are desired. As mentioned earlier, a user may send events to the server logic to indicate an interest to subscribe or publish, such requests will modify the users Entry Server behavior, to subscribe for particular areas on behalf of the client with the Communication Layer. Subsequent communications with the App Server (most likely) will be done through the client ↔ Entry Server ↔ Communication Layer ↔ App Server link.

F. Short Iterative Development

To reduce iteration time, we allow developers to make server-side code changes directly within a browser-based online editor, while the changes are immediately hot-reloaded as part of the currently executing functions. Often times, developing multi-user server applications require extensive testing and debugging over network and logic issues. Traditionally this process looks something like:

1) Make code changes locally
2) Compile and test code changes
3) Upload new / modified code to server
4) Shutdown server
5) Restart server
6) Test code changes with client

With ImonClouds online editor and hot code reload, developers are able to make code changes directly with the online editor, while the changes in script, because they can be parsed dynamically, the changes / updates are reflected immediately. The amount of time to iterate a given modify-verify test cycle thus can be reduced dramatically.

III. Current Progress

Currently we have implemented the logic execution environment, as well as the basic SPS operations at the Communication Layer. A prototype Entry Server is also available, though currently it connects directly with Lobby or App Servers, and does not yet support performing SPS operations with the Communication Layer.

Regarding scaling and deployment, ImonCloud currently can start/stop App Servers based on the number of connected users, though it does not yet support starting new servers at a different geography and performing code migration, as currently we only have deployment in one geographic area. We expect these features to become available as our deployed servers exist at different geographies.

Code migration to other App Servers to reduce communication time between users and their App Servers is not yet done, and as a first step we may simply replicate logic scripts at all the App Server, and only test for dynamic App Server startup to handle workload, before performing on-the-fly code migration.

IV. Conclusion

Developing, deploying and operating online games are no simple tasks for game developers. We design ImonCloud to facilitate this process by providing the spatial publish/subscribe (SPS) construct, so that players can receive relevant messages by only specifying spatial interests. To ease deployment, developers can place their code just once, and deploy globally using proximity execution, so that the game logic is executed at a server close to the player, reducing communication latencies. Finally, to ease operations, App Servers that execute game logic can always be dynamically adjusted, both in number and location. With just enough redundancy, the crashing of any single server will also not harm the normal operations of the game.

We are in the process of deploying the first version of ImonCloud by hosting of a commercial game. However, as the initial deployment is limited in one geography, we may only evaluate "proximity execution" to a limited degree. Evaluating the usage of SPS in ImonCloud will be carried out by some experiments simulating a large number of player entities. Also on the plans are to work closely with game developers on the usage and adoption of ImonCloud, so that its benefits, limitations can be more visibly found and demonstrated.

References

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

[2] 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.

[3] Y.-T. Lee and K.-T. Chen, "Is server consolidation beneficial to mmorpg? a case study of world of warcraft," in *Proc. 2010 IEEE 3rd International Conference on Cloud Computing*. ACM, 2010, pp. 435–442.

[4] J. Waldo, "Scaling in games & virtual worlds," *ACM Queue*, vol. 51, no. 8, 2008.

[5] M. Tayarani Najaran and C. Krasic, "Scaling online games with adaptive interest management in the cloud," ser. NetGames '10. Piscataway, NJ, USA: IEEE Press, 2010, pp. 9:1–9:6.

[6] S.-Y. Hu and K.-T. Chen, "Vso: Self-organizing spatial publish subscribe," in *Proc. Fifth IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2011)*, 2011.

[7] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, "Vivaldi: a decentralized network coordinate system," in *Proc. SIGCOMM*, 2004.