ABSTRACT
Massively multiplayer online games (MMOGs), which are typically supported by large distributed systems, require a scalable, low-latency messaging middleware that supports the location-based semantics and the loosely coupled interaction of multiplayer games components. In this experimentation paper, we present three different pub/sub-driven designs for a MMOG networking engine that account for the highly interactive and massive nature of these games. Each design uses not only different pub/sub approaches (from topic-based to content-based) but also serves varying degrees of responsibilities. In particular, some of them integrate game functionality, such as interest management, into the network engine. We implement, evaluate, and compare our proposed designs in the MMOG prototype Mammoth. Our real-world results show the viability of pub/sub while at the same time highlighting clear trade-offs between the different designs used, especially in the number and frequency of the various message types, such as subscriptions.

Keywords
Publish/Subscribe, Multiplayer Games, Interest Management

1. INTRODUCTION
Massively multiplayer online games (MMOGs) have become tremendously popular in the past decade [15]. The fast growth of the genre has greatly expanded the massive scale of the games, which puts into question the notion of the centralized client/server architecture used in older multiplayer games. Traditionally, the game state is managed by a central server which distributes replicas of in-game objects to the clients. Each player receives objects that he/she needs to be able to play (e.g., objects located close enough to their in-game avatar). This set of interesting objects is determined by an interest management module that runs on the server. Clients interact with the world by sending actions to the server. Changes to the game state are then sent as updates to the replicas of the modified objects.

Distributed game architectures were developed to address the scalability problem posed by MMOGs [4][3][2]. Instead of being managed by a centralized server, the objects are now divided onto multiple peers; as a result, clients receive updates for objects from multiple sources each of them following a one-to-many communication pattern. Furthermore, ownership of objects may change for load-balancing purposes. Interest management itself can also be distributed among many servers.

These loosely coupled interactions suggest that a pub/sub-driven design constitutes a good fit for a MMOG communication fabric, and in fact, pub/sub concepts can be found in many game engines. In this experimentation paper, we propose three network engines that employ different pub/sub matching semantics (from topic-based to content-based) to support the various interaction mechanisms required by MMOGs. In particular, we go beyond traditional gaming use cases for pub/sub and show how its filtering capabilities (particularly location-based filtering) can be used to offload responsibilities from the game servers, such as interest management. Each of the engines uses a different approach to map game tasks to subscriptions and publications, leading to interesting trade-offs in terms of the number of messages needed for the various tasks and game scenarios.

This experimentation paper offers the following contributions:
• We present three network engine designs that adapt topic-based and content-based pub/sub technology for use in MMOGs.
• We demonstrate a novel application of pub/sub as a decentralized interest manager.
• We have fully integrated our network engines into Mammoth [13], a MMOG research framework showing the feasibility of our approaches.
• We show extensive performance experiments using our Mammoth prototype that show the advantages and disadvantages of each network engine.

The remainder of this paper is structured as follows. Section 2 shows basic concepts for MMOGs, with both the traditional client/server and our distributed pub/sub architectures. Section 2.1 describes our game architecture and its requirements. Section 3 presents some of the related work. Section 4 presents our pub/sub network engines, followed by Section 5 containing our experimental evaluation. Finally,
2. BACKGROUND

This section provides an overview of the general architecture and concepts used in MMOGs and in pub/sub systems.

Every machine in a game network is known as a node. There are three main types of nodes: servers, clients and brokers. Servers handle most of the game services and handle incoming connections. Clients are run by players that join the game. Every client controls an avatar inside the game, which is used to interact with the game world by performing actions, such as moving, picking up objects, and interacting with other players. Clients also provide a user interface, which presents a graphical rendition of the game world, also called map, to the player. Finally, brokers are specialized nodes used by the pub/sub system to route messages between machines.

2.1 Pub/Sub Architecture

Our MMOG architecture employs a pub/sub layer operating on a broker overlay network. This particular dissemination model allows data sources (publishers) to publish data to a pub/sub broker, which forwards the publication data to interested consumers. Consumers express their interest by submitting subscriptions, which are matched against publications by the pub/sub system. We employ two types of subscription languages. Topic-based pub/sub allows subscribers to subscribe to particular topics and receive all publications sent on the topic channel. Content-based pub/sub enriches the language by allowing predicates to be specified, which further filter publications containing additional attribute-value pairs in addition to the topic.

The pub/sub system is the core of the networking engine used to send messages between game nodes. In particular, we employ the multicast and filtering capabilities when publishing data that needs to be delivered to multiple subscribers. One common example is to disseminate game state update events to clients. For instance, a character that is moving needs to send its updated location to all clients who are within the visible range of the character. However, our work goes beyond standard use cases and uses pub/sub to support mechanisms typically reserved for other game components. We discuss this in detail Section 4.

2.2 Replica Management

 MMOGs follow a primary copy replication scheme. Each object in a game has a master copy, which is considered the authoritative state of the object. The node holding the master copy is considered the master holder for that object. Other nodes interested in a object can obtain a replica copy from the master-holder. In principle, any node in the system can hold master copies. Everybody else interested in the object must hold a replica. If a client wants to make an update to an object and it has the master copy, it can simply submit the change to the local master copy where it is executed.

If the client only has a replica, the replica must forward the request to the master copy. The replica must therefore know the location of the master holder and must be notified of any change in its location. Finally, after the master copy has been updated, the change is asynchronously propagated to the nodes that hold the replicas.

In standard client/server architectures it is typically the server(s) that hold the master copies. In our Mammoth framework, master copies can be located on any node, which allows us to distribute master copies among clients alleviating the load of the servers. This is facilitated by using a pub/sub infrastructure for update dissemination. In this paper, we take advantage of this, and assume that master copies reside on client nodes. We will see later that this allows for two of our pub/sub architectures to not require server nodes at all. In these cases, the game functionality, at least the one discussed in this paper, is fully distributed among clients and brokers.

2.3 Interest Management Overview

Interest Management (IM) is the mechanism that determines which in-game objects in the game map are interesting for each node in the system. In this context, interesting means that those are the objects for which the node must have information (i.e., a replica copy) in order to properly handle its duties. For example, a player node must be aware of all in-game objects that are inside its avatar’s sight range to be able to display them on the player’s screen. The area inside which in-game objects are deemed to be interesting is known as the Area of Interest (AoI). Research in IM techniques has focused on reducing its running time complexity and correctness (i.e., avoid false positives and negatives) [5]. For the purpose of this paper, we study two different IM mechanisms: rectangle-based and tile-based.

In rectangle-based IM (Fig. 1 on the left), a node’s AoI is determined by a rectangle centered over a point. For a player node, the rectangle is centered over its avatar. Another player is interesting if its location is within that rectangle. In order to calculate which objects are interesting, the node running the IM task (called the interest manager) must compare the positions of all objects in the game with the AoI of a particular game object to determine if they are interesting. Rectangle-based IM considers only the players’ location, and not properties of the world. For example, as shown on the right side of Figure 1, it can conclude that objects behind a wall are interesting, and thus require a client to obtain replicas for them, even if the player cannot interact with or see them on screen, since they are behind the wall. We consider such a case to be a false positive match. False positives cause a client to hold more replicas than necessary.

The tile-based IM (Fig. 1 on the right) requires the world map to be divided into smaller sub-sections called tiles. These tiles are generated to be obstacle-aware, such that tile borders will never cross impassible boundaries. One way to achieve this is through a constrained Delaunay triangulation of the world map [17]. The interest manager can leverage the tiles to approximate the AoI of an object. Determin-

\footnote{Some optimizations might be possible to not perform an all-against-all comparison.}

Figure 1: Rectangular (left) and Triangle-Based (right) IM
ing interest is then reduced to verifying if a target object is located on a tile which is contained within the AoI of the source object. Computing this set of tiles is performed by drawing a rectangle around the source object. Tiles are then considered in the AoI if they intersect with the rectangle and are not behind an obstacle.

The size of tiles is a parameter that can be adjusted: using bigger tiles decreases the number of tiles that are part of an AoI of a given size, while smaller tiles will increase the precision of the rectangular AoI approximation.

Usually the tile structure is determined before the game starts and does not change. Thus, it is possible to pre-compute the AoI for each tile. Then, after the game has started, the pre-computed AoI can be easily obtained for any player that is located inside a particular tile.

To perform IM efficiently, each tile can have pointers to the objects (including players) that are located on the tile. Then, when a player moves from one tile to another, IM can quickly determine the new objects the player is interested in. As the AoI shifts when a player enters a new tile, IM can simply look up the new tiles that were not part of the previous AoI, and determine that all objects in these new tiles are now interesting for the player. Although IM can be distributed across multiple nodes, each one managing different game world sections, the IM nodes must be subscribed to a relatively large amount of objects in order to perform their duties, which may create a bottleneck in the system.

In this paper, we propose a way to allow each client perform its own IM by taking advantage of the services offered by pub/sub engines. In this way, we can remove specialized IM nodes from the system entirely.

2.4 Network Communication

Message exchange in computer games is partially based on point-to-point communication, partially on multicast. For example, when a client joins a game, initial startup communication messages provide the client with the current game state. Other examples of communication are coordination messages among game servers, or game services such as chat. There are two tasks, however, that occur very frequently, and dominate the bandwidth requirements:

- **Update Messages:** When a client wants to update the state of a game object or avatar, an update request has to be sent to the master holder. This typically involves point-to-point communication from the requesting node to the master holder. After a master copy is updated, the master holder must send the updated game state to all nodes holding replicas. This is a multicast communication. As each update request message leads to many update messages, the latter is the dominating message type. To limit the size of the update, the master only sends changes since the last version (called delta updates) instead of the complete state of the object. This technique allows existing replicas on a client to stay up-to-date, but does not provide enough information to create a new replica, if needed.

- **Replica Transfers:** Once a client discovers a new object, it must obtain a replica of it. First, the master holder must be notified that the client needs the replica using a Replica Request Message, then the actual transfer to the client who needs the copy is performed. In a classical implementation both steps use point-to-point communication. As the request message is typically much smaller than the message containing the entire replica, the latter will be the dominating message type.

In this paper, our pub/sub engines are used for both point-to-point and multicast communication. The latter is a straightforward fit for a pub/sub engine. To emulate point-to-point communication, on the other hand, we designed and implemented different pub/sub approaches, each one with its own advantages as well as disadvantages. We outline them in detail in Section 4.

3. RELATED WORK

While there exists a huge variety of papers that study all kinds of publish/subscribe systems [10], very little of that work deals directly with multiplayer games and their specific requirements. In particular, to the best of our knowledge, no work employs a publish/subscribe system to distribute not only updates but also replicas to the players as we do.

Mercury [4] is a decentralized content-based pub/sub system. The core of the system is similar to the rendezvous concept shown in Scribe [6] or Hermes [16]. Subscriptions and publications are sent to a rendezvous point where they are matched and then delivered to their recipients. Publications are represented as lists of attribute-value pairs that contain game-related information, like a player’s position. Conversely, subscriptions consist of predicates that use operators to try and match a particular attribute using its content-based engine. Nodes are partitioned into circular groups, each group in charge of a specific attribute, while the internal nodes are in charge of a particular range of the attribute. This has the effect of distributing the network load across the nodes, reducing hotspots. Client nodes can then subscribe to the attributes that they consider interesting (for example, subscribe to all messages published in a radius around its avatar). This kind of architecture is indeed very powerful for small-scale games up to several tens of players where a client can have replicas of all objects in the game and only needs to subscribe to their updates, like in a first-person shooter (FPS) game.

G-COPSS [7] proposes a content-oriented pub/sub system that partitions the game world in a hierarchical way. Maps are divided into arbitrary layers, where each layer can be further divided into more. Therefore, a player can send or receive updates to the particular layers of the map he’s interested in. By subscribing to a particular hierarchical layer, a client can control its area of interest and the amount of messages it receives. As with Mercury, G-COPSS doesn’t consider replica transfer, so clients must have a replica of all in-game objects.

The previous two systems are able to handle the distribution of updates across a game using a pub/sub infrastructure but are missing one critical factor that limits their usefulness in MMO games: they depend on other mechanisms to distribute replicas to the client nodes. In the case of small-scale games it is possible for all clients to simply create a replica for each player/object in the game at startup. But in an MMOV, the sheer amount of players/objects makes it unrealistic for a client to hold a replica of each one. Therefore, an interest management mechanism must be put in place. Our network engines are not only able to distribute update messages but also the replicas themselves, and perform interest management, which is one of the main contributions of this paper.

Beyond multiplayer games, our replica distribution use case is a novel usage of publish/subscribe technology. There
exists a variety of location-based applications which use pub/sub to propagate publications to nearby elements [11]. However, those event-based applications typically employ publications as self-contained units of data which are thus consumed individually. In our case, those publications are used as updates to replicated objects which must be disseminated to subscribers at the time they emit the subscription. To seamlessly distribute the replicas, the pub/sub system would have to maintain updated copies of each object and deliver replicas to individual subscribers as they join the system. The closest mechanism that resembles this interaction found in literature is historical queries [14], which allow subscriptions to retrieve past publications. In theory, this would allow a newly joined subscriber to retrieve all the publications to date for a particular object and therefore reconstruct the latest state. In practice, online game workloads contain a large volume of publications which overwrite the same attributes (i.e. position updates). Therefore, it is inefficient to retrieve the complete log of publications when only a subset of them are reflected in the latest state of the object.

An alternative possibility is to use the pub/sub system to notify clients that objects have been updated, without actually delivering the content of the update. Thialfi [1] is one such notification system designed to notify clients that Google items, such as calendars, have been updated and need to be resynchronized by the client. A game could leverage such a system to notify clients that replicas they are interested in have been updated, and rely on the clients to retrieve the latest state of objects when needed. The advantage of this approach is that subscribers no longer need to receive every single publication, since they can retrieve the changes on their own when needed. This pull approach, however, needs to be complemented with a mechanism for clients to efficiently retrieve the relevant state updates.

4. PROPOSED PUB/SUB ENGINES

In this section, we propose three network engine designs for MMOGs. Two of them are based on a topic-based pub/sub system, while the third uses content-based pub/sub.

Use Cases: To highlight the differences in their respective design, we detail the functionalities of the engines along three basic use cases. These use cases pinpoint how each engine requires the use of a separate IM service, which is run on one or several IM server nodes. The IM service receives the movement updates of all players, and constantly monitors the interest of each player. When an IM server has determined that a player is newly interested in an object, the player’s client subscribes to the object’s topic. When the master copy of an object is updated, the master holder publishes the update message on the object’s channel.

4.1 Object-Based Network Engine

Our first network engine design’s core idea is to employ a topic-based pub/sub system and associate one topic with every object in the system. This creates a separate communication channel for each object: interest in an object is made explicit via a subscription to the related topic.

4.1.1 Basic Network Engine Description

Unlike the other pub/sub engines that will be introduced in the following subsections, the object-based network engine requires the use of a separate IM service, which is run on one or several IM server nodes. The IM service receives the movement updates of all players, and constantly monitors the interest of each player. When an IM server has determined that a player is newly interested in an object, the player’s client subscribes to the object’s topic. When the master copy of an object is updated, the master holder publishes the update message on the object’s channel.

4.1.2 Use Cases

• Object Discovery: This task is entirely done by the IM service, which keeps track of all player character and objects located in the game, and determines when an object becomes interesting to a player.

• Replica Transfer: When a new object becomes interesting to a client node because the avatar moved closer to the object or vice versa, the IM service determines which node holds that object’s master. It then sends a point-to-point message to the client via the client’s topic to inform it that it needs the object. The client subscribes to the object’s topic to receive future updates, and also sends a Replica Request Message to the master node via the master’s topic. Likewise, the IM service also determines when an object stops being interesting to a client. When this happens, the IM sends a message to the client via the client’s topic, and ask it to drop its replica and unsubscribe from the object’s topic. The replica transfer...
procedure is illustrated in Figure 2. In this example, client A and B are both master holder of their own players P1 and P2 respectively. The IM server has replicas of both. When the IM server determines that P1 enters P2’s AoI, it sends a message to B (1). B then subscribes to P1’s channel (2) and also sends a replica request message to node A (5). A returns the replica to B (4).

- **Update Propagation**: When a client wishes to update the status of an object and it does not have the master, it must first send a point-to-point message with the update request to the master holder using the master holder topic as seen in Figure 3 (1). In the figure, client A controls avatar P2 but only has a replica of it while client B has the master copy. The master-holder then determines if the update is valid and sends two messages: a confirmation message that is sent back to the requesting client (2), and an update message containing the state change which is published using the object’s topic. The update is thus disseminated to all the nodes that are interested in (and thus have subscribed to) that particular object/topic (3).

4.1.3 Discussion
The only message type that takes advantage of the multicast capabilities of the pub/sub infrastructure is the update message. All other messages are sent point-to-point. A node subscribes to its own channel only once, when it connects to the system. Subscriptions and unsubscriptions to/from object channels occur whenever the interest set of a player changes.

4.2 Tile-Based Network Engine
The tile-based engine can be used with game engines that provide a partitioning of the game map into tiles as discussed in Section 2.3. We assume a triangular world partitioning that follows the borders of immutable objects such as walls, trees or other obstacles as described in Section 2.3. In this engine, most of the communication between nodes is done via topics linked to map tiles. Another important characteristic is that this engine does not require a dedicated IM service to be present. Instead, tile-based IM as described in Section 2 is integrated into the pub/sub-based communication.

4.2.1 Basic Network Engine Description
Additionally to the node topics that facilitate basic point-to-point communication, the tile-based engine creates three different topic channels for each tile in the game world, as illustrated in Figure 4:

- **Replica Request**: This topic is used to request replicas. Master holders subscribe to the Replica Request topic of a tile if at least one of their master objects is located in it. Thus, when a node wishes to obtain the replicas of the objects located in a particular tile \( T \), it can use the Replica Request topic of \( T \) to contact all nodes that hold master copies of objects located in \( T \) with a single publication.
- **Replica Reply**: These topics are used to transfer the actual replicas. Before a node makes a request for replicas in a particular tile \( T \) (through the Replica Request topic), it subscribes temporally to the Replica Reply topic of this same tile \( T \). When a master holder receives a replica request through the Replica request topic of a tile \( T \) it publishes a replica of each of the master copies it has that are located in \( T \) on the Replica Reply topic of \( T \). Subscriptions to Replica Reply channels are temporary, and the nodes unsubscribe after enough time has been given for replies to arrive.
- **Notify Update**: Nodes subscribe to the Notify Update topic of the tiles that lie in the AoI of their avatar. As this AoI is pre-computed for each tile at the beginning of the game, each client knows immediately to which new Notify Update topics to subscribe to when moving from tile \( T1 \) to tile \( T2 \), the Notify Update topics of tiles that are not in the AoI of \( T1 \) but in the AoI of \( T2 \).

4.2.2 Use Cases
- **Update Propagation**: As mentioned above, clients subscribe to the Notify Update topic of all tiles that lie in their AoI. When a player wants to update an object (e.g., its avatar), it sends an update request to the master holder via the master holder’s topic. The master holder performs the update and multicasts the change using the Notify Update topic of the tile the object resides on. Thus, everybody interested in this update receives it through this publication. Conceptually, this is similar to what is done in the object-based approach, but instead of publishing the change on the object’s channel, the master holder publishes it on the Notify Update channel of the tile the object resides on. In order to always only receive relevant state updates, a player that moves has to subscribe to the Notify Update topic of tiles that enter its AoI and unsubscribe from the ones that are not in its AoI anymore.
- **Object Discovery**: The tile-based engine does not require a dedicated IM service to solve this task. Object discovery is done through the pub/sub system. To explain object discovery and replica transfer, we will use the following example situation. Imagine a player \( P1 \) that moves to a new tile \( T \), and a stationary player \( P2 \) that does not move. Before \( P1 \)’s move the two players were not interested in each other but, after the move they are, i.e., \( P1 \) is now in a tile that belongs to the AoI of \( P2 \) and vice versa. In this situation, a) \( P1 \) has to discover all objects, including \( P2 \), that are in its new AoI, and b) \( P2 \) has to discover that \( P1 \) moved into its AoI.
To achieve a), master holders always subscribe to the Replica Request topic of the tile in which their game object
is located. Knowing this, the client of the moving player P1 first subscribes temporarily to the Replica Request topics of all tiles that are newly in its Aol, and then publishes a Replica Request message on the Replica Request topics of the same set of tiles. Note that this message does not mention any particular game objects, because P1 wants to have replicas of all objects residing in these tiles. All master holders that hold game objects that reside in these tiles receive the publication, and hence discover that P1 needs replicas of the master object they are holding. Achieving b) is straightforward. When P2 moves to tile T, the master holder of P2 publishes the location change on the Notify Update topic of tile T to propagate the update to all interested nodes. Since Client A has already subscribed to the Notify Update topic of T because it belongs to P1’s Aol, it receives this update publication. It then discovers that it does not yet have a copy of P2.

- Replica Transfer:
  Replica transfer is straightforward. If it was a master holder that discovered that a client needs a replica as described above in a) where P1 sends the Replica Request message on the new tiles, the master simply sends the replica on the Replica Reply topic, and P1 will receive it. If it is the client that discovered that it needs a replica because it received an update for an unknown object as described above for player P2 in situation b), the client temporarily subscribes to the Replica Reply topic of the corresponding tile. It then also publishes a request to receive replica of that specific object on the Replica Request topic of the corresponding tile. The master holder of the object receives the request as it has subscribed to the Replica Request topic of the tile, and sends the replica on the Replica Reply topic.

The protocol for object discovery and replica transfer in case a) is illustrated in Figure 5. In the figure, P1 is the avatar of client A. It subscribes to every new tile’s Replica Reply channel (1) and publishes a general replica request message on the corresponding Replica Request channel. Clients B and C have master copies of objects P2 and P3 residing on this new tile. Thus, they send replicas of these objects on the Replica Reply channel of the tile. The protocol for case (b) is depicted in Figure 6. P2 is the avatar of client B, which has received the update message of P1 (1). It subscribes to tile T’s Replica Reply topic and publishes a request to receive specifically a replica of P1 on T’s Replica Request topic (2). Client A is the master holder of its own avatar P1. It will receive the request and send the replica on T’s Replica Reply topic. Note that the client of an avatar does not necessarily hold the primary copy of the avatar. We only did this in this example to simplify the figures.

4.2.3 Discussion

While update propagation is conceptually similar to the object-based engine, with the only difference being that the update is published on a tile and not an object, object discovery and replica transfer are very different, and do not contain any point-to-point communication anymore. For instance, instead of requesting a copy from the master holder, the request is published on the Request Replica topic of a tile. In the example above, this reduces the messages P1 might have to send. For each new tile that becomes interesting, it only sends a single message that requests all objects that reside on this tile. This message is received by all nodes that have master copies of objects located on this tile. If there are several objects on the same tile, only a single request message needs to be sent. Also, instead of sending the replicas directly back to the requesting nodes, they are published on a channel. If several players request objects of a tile concurrently (which can happen when they move as a group), the master holders receive several requests at around the same time. In this case, they publish each replica only once on the Replica Reply channel, instead of sending them individually to each requestor. How this replica transfer strategy affects performance in certain game scenarios is illustrated in the evaluation section.

4.3 Area-Based Network Engine

Content-based pub/sub engines are much more expressive than topic based engines in terms of the type of subscriptions and publications, as subscriptions can contain filters allowing for much more fine-grained multicast opportunities. MMOGs can take advantage of this additional functionality. In particular, we can use them to specify locations and Aol ranges. More specifically, publications about game events can be accompanied by two attributes, the x- and a y-coordinates of the game world in which the event takes place. That is, when a player moves, it simply needs to publish its new location, which is a specific x, y-coordinate, indicating these x, y-coordinates as the values for the x- and y-attributes. Furthermore, in order to declare a rectangular Aol, one only needs to subscribe, using predicates, to the x- and y-ranges that define the rectangle. Thus, a player movement will be received by all clients who have subscribed to a Aol rectangle which includes the x/y coordinates of the location the player moved to.

In the following, we define the details of the area-based network engine, which uses this idea. Julie like the tile-based engine, the area-based engine incorporates IM, and hence no extra IM service is needed.

4.3.1 Basic Network Engine Description

Similar to the tile-based approach, the content-based network engine distinguishes between three different specialized
topics to deal with the different use cases. The topics are the same as in the tile-based engine, as illustrated in Figure 7. The main difference is that there are only three topics for the entire world (as opposed to three topics per tile):

- **Notify Update**: Each client node subscribes to a Notify Update topic with x and y ranges that correspond to the AoI rectangle centred around the avatar. An update to an object is published on a point (x/y-coordinate) via this topic. If the point is within the AoI rectangle of a client’s avatar, the client receives the update.

- **Replica Request**: The master holder of an object subscribes to a AoI rectangle on the Replica Request topic that is centered on the object’s position. When someone needs a copy, it has to publish a request on the topic indicating any point that resides within this rectangle, and the master-holder will receive the request.

- **Replica Reply**: Finally, in order to receive replies to their requests, a client node subscribes temporarily to the L-shaped area that represents the new area that it wants to receive new replicas for by issuing two rectangular subscriptions on the Replica Reply topic. The master holder has to publish the replica on this topic using a point that resides within one of these two rectangles for the client to receive it.

### 4.3.2 Use Cases

The use cases are handled in a similar way than in the tile-based engine except for some subtle differences.

- **Update Propagation**: Update propagation is conceptually the same as in the tile-based engine. When a player wants to update an object, it sends an update request to the master holder via the master holder’s topic (point-to-point communication). The master holder performs the update and publishes the changes on the Notify Update topic using the current x/y coordinates of the object. Any client whose avatar’s AoI contains that point will receive the update as they have to subscribed to that area of the Notify Update topic.

- **Object Discovery**: Just like for the tile-based engine, we distinguish between discovery of interesting objects for a player that moves (case a), and discovery of objects that enter the AoI of a stationary player (case b). For case a) it first must be noted that the frequency of adjusting the AoI of a moving player is configurable. One could do it continuously, but this would lead to many AoI subscriptions and possibly many replica publications. Thus, it makes more sense to redefine the AoI whenever a certain time interval has passed or the player has travelled a certain distance. At this point, the client of the moving player subscribes to L-shaped region that covers the newly discovered territory on the Replica Reply topic. Next, the client publishes on the Replica Request topic using the current location of its avatar as point. All master holders whose Replica Request areas contain the avatar’s location will receive the request and hence discover that a replica needs to be sent.

Case b) is a lot simpler, because the client of the stationary player receives a position update on the Notify Update topic for the moving player, but it does not have a replica for it.

- **Replica Transfer**: Replica transfer is again straightforward. If it was a master holder that discovered that a client needs a replica because it received a Replica Request message as described above in a), the master simply publishes a replica on the Replica Reply topic using the master copy’s location as point. If a master holder receives several replica requests concurrently, it only sends out one replica to reduce the replica transfer overhead. Thanks to pub/sub, all clients who have subscribed to a Replica Reply area that contains the object’s location, will receive the replica. If it is the client that discovered that it needs a replica because it received an update for an unknown object as described above in situation b), it subscribes to a Replica Reply topic that contains the position of the new player it detected, and then publishes a Replica Request message on the point of this new player so that its master holder receives it.

### 4.3.3 Discussion

Conceptually, tile-based and content-based engines are similar. The main difference lies in how the AoI is defined. In the tile-based engine it consists of a set of tiles, while in the area-based one it consists of a single rectangle. This has consequences when the AoI changes. In the tile-based approach, the client subscribes to all new tiles and unsubscribes to all tiles that are no more in the AoI. Depending on the size of tiles vs. size of AoI, there can be many such tiles leading to many subscriptions and unsubscriptions for the Notify Update and Replica Reply topics. In contrast, using the area-based engine, the avatar only needs to issue one subscription for the Notify Update topic, and two for the Replica Reply topic (due to the L-Shape of the area). For the replica request publications, the client sends one publication for each new tile in the tile-based approach vs. a single one in the area-based approach. Thus, the area-based approach uses overall less messages than the tile-based approach. However, in the area-based approach, the area for which replicas are requested is rectangular, and hence is not obstacle-aware.

### 5. EVALUATION

To evaluate our network engines we implemented them in Mammoth, a massively multiplayer research framework [13], which allows us to run real-world experiments (as opposed to simulations).

In this evaluation section we present three sets of experiments. The first set of experiments aims at demonstrating that all three network engine designs scale when more brokers and clients are added to the system. The second set of experiments is intended to show the behavioural differences of the three network engine designs by running scenarios typically encountered in MMOGs and comparing gathered metrics, e.g., the number of subscriptions, publications, update messages, etc. The last set of experiments illustrates how to determine the best partitioning of the virtual world in order to optimize the performance of the network engine that is based on tiles.

As pub/sub engine we used PADRES [12], which supports
both content- and topic-based subscriptions and offers various broker configurations.

### 5.1 Experimental Setup

In all experiments, our pub/sub brokers (and interest managers, in the case of the object-based network engine) are run on our dedicated laboratory machines (dual-core processors, 3.06 GHz, 4GB memory, 2 network interfaces, using Linux). The clients are executed on the computers available in the student labs of the School of Computer Science of McGill University (dual-core processors, at least 2GBs memory, Linux). The first ten percent of the clients that join the system register as master-holders, and as a result all the master objects in the game are evenly distributed to them during start-up.

During our experiments we also run a special Mammoth node that provides a user interface that can send remote commands to all connected clients. In this way, the clients can be instructed to follow the behavior required for a particular experiment. Clients can also be given an AI script that is executed as soon as they join the system. For instance, they can be told to move their avatar to a random location every ten seconds or to follow a pre-defined path. In order to provide for a fair comparison between engines, we configured the content-based engine such that players change their AoI subscriptions with the same frequency as in the tile-based approach. To do so, we take advantage of the fact that maps in Mammoth are always triangulized. More specifically, in the content-based engine, a player subscribes to a new AoI and requests accordingly replicas in the L-shape area that defines the new part of its AoI whenever it moves from one tile to another in the Mammoth map. Furthermore, in all experiments, players send update messages in fixed intervals of 1 second. That is, if they have moved within the last second, they send one update message with their new position (and no message if they haven’t moved).

### 5.2 Scalability Experiments

#### 5.2.1 Experimental Setting

To demonstrate the scalability of our network engines, we ran a set of experiments where the number of clients on our system is slowly increased until our main performance metric, the average response time of player actions, raises above a threshold of 400ms, which is considered unacceptable for a fast-paced game [8][9].

The experiments were conducted on the Mammoth map "Town20-500", a virtual environment representative of MMOGs, shown in Figure [ref]. This map has thousand available avatars to be controlled by the clients.

In all experiments, the brokers (and servers performing interest management in the case of the object-based engine) are started first. Then, new clients join the game by connecting to a random broker every eight seconds. Once connected, they take control of an avatar and start wandering randomly around the world.

We ran the experiments using one, two, four, and eight fully connected PADRES brokers. For each experiment, we measured several main data points. One important is the **response time** (as measured by the RPC round-trip time) perceived by the clients: it consists of the elapsed time between when the client sends a remote move message to the node that holds the master copy of the avatar it is controlling and the time at which it receives the confirmation of this action from the master holder. We also report on the network traffic on the brokers, as well as the broker’s CPU and memory usage.

#### 5.2.2 Experimental Results

The results of this set of experiments show that all three proposed network engine designs scale well. Figures 8a to 8c show for each of the network designs and for different numbers of brokers how many players can be supported while keeping the average response time below 50, 100, 200, and 400 ms. Response times above 400 ms are not acceptable if a good user experience is to be maintained.

The first insight that can be extracted is that all three designs perform similarly when only 1 broker is used: 170 players can easily be supported (response times below 50ms), and the maximum of players that can be handled is around 230 (response time reaches 400ms).

The experiment also shows that all network engines benefit from adding more brokers to the system, but the increase in performance is not linear. Each extra broker improves the performance of the system by a smaller amount than the previous one. It also shows that the scalability trend of the tile and the area-based engines are extremely similar. They both benefit more from additional brokers than the object-based network engine. For example, with 8 brokers, they can support up to 900 players, whereas the object-based engine can only handle 750.

The reason behind this is that all messages in the object-based engine are sent point-to-point. As a result, an update that has to reach n interested players results in n different update messages that need to be sent through the brokers. Meanwhile, the tile and area-based engines take advantage of the client’s subscriptions to a tile or area to multicast the update messages to all interested parties. Thus, an update that has to reach n clients requires just a single update message to be sent to the brokers, reducing their incoming message load. Of course, in all 3 designs the end-point brokers must still send individual messages to all the interested clients.

To determine the bottleneck(s) of the system we show in Figures 9a-9c detailed measurements of the 1, 2, 4 and 8 broker experiment for the object-based network engine. The details for the other engines are not shown for space reasons but show similar behavior.

In Figure 9a we see that with 1 broker, the system is able to support up to 230 clients before the average response time deteriorates to 400 ms or more. With 2 brokers, 350 clients can be supported. The 4 broker configuration can handle close to 600 clients, whereas the 8 broker configuration supports 750. The figure also shows that the base response time measured for low numbers of clients increases for configurations with higher number of brokers. With 50 connected clients, the 1 broker configuration has an average response time of 25ms, whereas the average response time for the 8 broker configuration for 50 clients is close to 45ms. This is due to the fact that in the 8 broker experiment, on average 7 out of 8 messages have to be routed through 2 brokers, i.e., need 3 hops before they reach their destination. Thus, when the system is not heavily used, additional brokers lead to higher delays, while at higher loads, the increased power easily compensates for the additional delay.

The outgoing network traffic on the brokers increases al-
most linearly with each new client that connects to the system as shown in Figure 9b. The CPU usage, presented in Figure 9c, shows a similar trend. By comparing the results of this and other experiments we noticed that the client response time starts deteriorating drastically when the average output traffic on the brokers surpasses 16 MB/s. At that same moment, the broker’s message output queue begins to grow until it fills up and it can no longer send all the messages it is required to. Therefore, the broker’s output queue is the bottleneck in our system.

In all experiments, the average CPU usage at the broker at that moment is approximately 60%. Even when adding more clients after this point, the CPU usage of the overloaded brokers does not increase further, as shown in Figure 9c. It can therefore be concluded that the CPU is not the bottleneck.

Finally, we also measured the input and output network traffic and CPU usage on the client nodes (in particular the master-holders), but their values are much lower than the ones at the brokers.

5.3 Analyzing Behavioural Differences in Various Game Situations

5.3.1 Experimental Setting

The purpose of this experiment is to show the different behaviours that our network engines exhibit when exposed to typical MMOG situations. The scalability experiments of the previous section used a virtual world where the avatars were distributed uniformly, and were moving randomly around the map. In this experiment we set up controlled game situations with a smaller amount of players, and report on the number and types of messages produced by our different network engine designs.

The virtual world used in this experiment is the map "Town30-1000", a simple map with 1000 tiles and 30 available avatars, which are initially uniformly distributed across the map. The map also contains 1000 static game objects that are uniformly distributed across the map, i.e., roughly one game object per tile. The experiments use three brokers (and the corresponding three interest management servers, for the object-based engine). The player machines, of which 3 were configured as master holders, connect to a random broker.

After all the players have joined the game, they are remote-controlled to perform the following movements:
- Flock towards a given point A.
- After everybody has reached point A, the players wait until the 40 seconds mark and then move together towards a given point B.
- Once everybody has reached point B, the players wait until the 70 seconds mark and then they slowly disperse using random movements.

This behaviour represents typical situations in MMOGs: a point of interest is discovered in the game world that attracts players (flocking to point A), or a group of players is pursuing the same goal / wants to reach the same destination / is following someone (group movement from A to B). Figure 10a shows the number of active players, i.e., players that are moving, over time during the experiment. In the beginning, all players are moving towards point A. As some players reach their destination, the number of moving players diminishes until nobody is moving at around 35 seconds into the experiment. All players are now gathered at point A. At 40 seconds, the players are instructed to move towards point B as a group. They reach point B approximately after 65 seconds. After 5 seconds of idle time, they start dispersing. In total, the experiment lasts 100 seconds.

In order to compare the different network engine designs we ran this experiment five times. The first run is used as a baseline experiment to compare with the subsequent runs, for which we varied a single parameter of the experiment to analyze how this parameter affects the observable network behaviour. We performed the following variations:
- Smaller Area of Interest (players are configured with an AoI that is half the size of the one of the baseline);
- Faster Avatars (players move twice as fast as the baseline);
- More Game Objects (each tile contains on average 2 times more game objects), and
- With Obstacles (a straight wall is introduced into the world, starting at point A and ending at point B).²

For each experiment, we recorded the following set of measurements (averaged over intervals of five seconds):
- Number of subscriptions (to master objects, nodes, as well as area/tiles);
- Number of publications due to movement updates;
- Number of publications due to replica transfers.

The most interesting results for the five experiments are shown in Figures 10b to 12c, which allow us to make the following observations:
- All graphs show that the message traffic is due to the activity of the players. When all players are inactive (between 35s and 40s, and between 65s and 70s), no messages of any kind are sent through the brokers.
- Figure 10b, which shows the number of update messages that are received each second on the clients, shows that the most update messages are received when there are lots of moving players that are close together. This is the case when the players move in a group from point A to point B, and initially at point B when they start dispersing. It also shows that the two obstacle-aware engines (the object-based and the tile-based one) can take advantage of the wall between A and B to reduce the number of update messages that need to be processed. Since the wall splits the group of moving players in two, the number of update messages at that time is reduced by a factor of 2.
- The graphs that show the number of subscription messages and replica transfer messages (Figures 11a to 12c) illustrate that there are less subscriptions and replica transfers when the players move as a group from point A to point B (between 40s and 65s). There are three reasons for this: First, when the players are all gathered in point A, they already have acquired replicas of all other avatars, since they are all in each other’s interest area. Hence, when they start moving as a group, they do not require new avatar replicas to be transferred from the master holders to the client node. The only replicas they need to acquire are the static game object replicas that they encounter in the map. Second, since the players are moving together as a group, they tend to become interested in the same set of tiles / same area as they move, so the master holders (each of which handles multiple avatars)

²Since tiles are central to the tile-based network engine, we dedicated a whole set of experiments to varying the tile size. These experiments are shown in subsection 5.4.
Figure 8: Average Client Response Time

Figure 9: Scalability

Figure 10: Active Players and resulting Update Message throughput for all engines

Figure 11: Number of Subscriptions

Figure 12: Number of Messages for Replica Transfer
only needs to subscribe to these new tiles / areas once. Finally, since clients become interested in new objects at about the same time, the master holders in the area- and tile-based engines need to send less replica multicast messages.

- Comparing Figure 11a and 11b shows us that the area-based engine requires less overall subscriptions than the tile-based one. This is easily explained: in the area-based design, a client, when moving from one tile to another, issues a total of 3 subscriptions: one subscription to the Notify Update Channel for a rectangle centred at his position that represents his interest range, and two subscriptions to the Replica Reply Channel for two rectangular areas that cover the L-shaped area for which the client needs to receive new replicas. Figure 11b shows that the number of subscriptions for the area based engine is independent of the area of interest, the number of game objects, and the presence or absence of obstacles. Only in the case where the avatars move faster, more subscriptions are issued.

In the tile-based one, on the other hand, a client must subscribe to the Notify Update Channels and the Replica Reply Channels of all new interesting tiles. The average number of new interesting tiles depends heavily on the tile size and the area of interest of a player. It does not depend so much on the density of game objects in the map. Even if there are multiple new game objects to discover within a tile, at most one subscription to the Replica Reply channel per tile is necessary. These results are confirmed when looking at Figure 11a.

- Figures 11a and 11b show that when avatars move twice as fast, the number of subscriptions issued by clients for the tile and area-based engines doubles accordingly. This is due to the fact that faster avatars cross tile boundaries more often, and hence their area of interest in these two engines changes more frequently. Figure 11c shows that the same is true for subscriptions issued by master holders in both the tile and the area-based engines. Additionally, as shown by Figures 12a to 12c, the number of replicas that need to be transmitted from the master holders to the clients increases, this time for all 3 engine designs. Note that the number of update messages, on the other hand, is the same for all engines. It is nevertheless affected from faster avatars as we have set the update dissemination interval for all experiments to 1 second. That is, as long as they move, they send the same number of messages as slower avatars. But as they move faster, they reach their targets faster. Once they have reached their target they do not send update messages any more until they move again (see Figure 10b).

- Figures 12a to 12c point out the fact that in the object-based engine the master holders send significantly more replica transfer messages to their clients. This is due to the fact that all messages are sent P2P. The tile and area-based engines can send a replica to several clients by means of a single publication on a tile or point in an area.

5.4 Analyzing Behavioural Differences when Varying Tile Size

5.4.1 Experimental Setting

The tile-based network engines take advantage of the partitioning of the virtual world into tiles to determine the AoI.

Figure 13: Number of Update Messages / Subscriptions and Total Outgoing Network Bandwidth

The purpose of this set of experiments is to analyze its behaviour when the tile size varies. We compare it with the area-based engine where the AoI is not dependent of the tile size. That is, in the area-based engine, the rectangular region that the client subscribes to in the update message channel corresponds exactly to the area of interest of the player. In contrast, to make sure that in the case of the tile-based engine the client receives at least all updates that happen within the area of interest of the player, the rectangular region is approximated by the set of tiles that intersect or are contained in that rectangle.

The experiment setup involves using a single broker, 10 master holder nodes, and 100 client nodes which control avatars that wander randomly through the world during the experiment. In total, we ran eight experiments on the same map with an increasingly more fine-grained, triangular partitioning. The first run used 8 tiles, the second one 16, then 32, 64, 128, 256, 512 and finally 1024 tiles.

Figure 13 shows the results for this experiment. First we can observe that the number of update messages received by the clients in the area-based engine is constant. In the tile-based engine the number of update messages is initially significantly higher, but eventually converges as the number of tiles increases. This is due to the fact that the rectangle approximation using triangular tiles in the tile-based engine largely over approximates the actual rectangular area of interest of the player when the tiles are big.

The figure also shows that the number of subscriptions in the tile-based engine increases significantly as the number of tiles increases. Therefore the ideal tile size for the tile-based engine depends on the weight of update messages compared to the weight of subscription messages. For the area-based engine, the number of subscriptions also increases. It is due to our specific setting where we decide to subscribe to a new AoI rectangle whenever the player enters a new tile. As the tiles become smaller and smaller, players reset their AoI more often.

In our engine, the bottleneck is network bandwidth. Update messages are 998 bytes, replica transfers are 3008 bytes, and subscription messages are 208 bytes long. Figure 13 also plots the total outgoing bandwidth used by the update and subscription messages combined. It shows that for our particular experimental setup, the bandwidth used is minimal when the map is partitioned into 256 tiles.

6. CONCLUSIONS

In this paper we presented three network engine designs
that adapt topic-based and content-based pub/sub technology for use in MMOGs. We showed that the one-to-many communication offered by pub/sub can be exploited to disseminate state updates and replicas efficiently to clients. Furthermore, we explained how interest management, a game service typically done by dedicated game servers, can be decentralized and performed by the pub/sub engine using clever subscription protocols.

We have fully integrated our network engines into Mammoth [13], a MMOG research framework, thus demonstrating the feasibility of our approaches. We ran extensive performance experiments that were executed on our Mammoth prototype with up to 1000 clients that show the advantages and disadvantages of each network engine.

In particular, our experiments showed that:

- The main factor for scalability is the limited network bandwidth on the brokers.
- All three pub/sub based engines scale. With the same number of brokers, however, the tile- and area-based engines outperform the object-based one due to the fact that they can take advantage of multicasting updates and replicas to multiple clients using a single publication.
- Player flocking and group movement drastically increases the number of update messages that are sent through the system. The built-in interest management of the tile-based engine can take advantage of obstacles to reduce that number. This is also the case for the object-based engine, if an obstacle-aware interest manager is used.
- The number of subscriptions required in the area based engine is significantly lower than the other engines. Additionally, it does not depend on the size of the area of interest of the avatars.
- While decreasing the tile size helps to reduce the number of update messages received in the tile-based engine, it simultaneously increases the number of necessary subscriptions. The ideal tile size therefore depends on the ratio between size of update message vs. size of subscription message.

To summarize, the tile- and area-based network engines are attractive network engine designs for MMOGs, in particular because they do not require dedicated game servers that perform interest management, and because they can take advantage of the one-to-many communication provided by pub/sub to disseminate game state and game state updates. The tile-based engine is most suited for game situations where there are lots of obstacles / corridors, particularly when there is a high likelihood of players flocking or moving in groups. The area-based engine is most suited for game situations where the area of interest of players is big and the world is open, i.e., it does not contain obstacles that separate players that are position-wise close to each other. In virtual worlds this is mostly the case when the player walks "outside".

While the focus of this paper is on MMOGs, we believe that our network engines can also be used for other location-based applications, e.g., traffic control systems, where external observers, but also the cars themselves want to see what is happening on the map. We are planning to investigate this in a near future. Furthermore, in order to improve the performance of the area-based engine, we are looking into ways of extending its interest management to be obstacle-aware. This might require extending the underlying content-based pub/sub engine.

7. REFERENCES