YOBOL: LOCALITY-AWARE MULTICAST ENGINE FOR A MASSIVELY MULTIPLAYER GAME ARCHITECTURE

by

Chhunry Pheng

School of Computer Science
McGill University, Montreal

October 2010

A THESIS SUBMITTED TO MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF
MASTER OF SCIENCE

Abstract

Massively multiplayer online games have gained such momentum throughout the years that their consumer base has exploded. Being mainly built on a client-server architecture, the server becomes the bottleneck, causing scalability problem. To alleviate this, peer-to-peer structures have been exploited in the game context.

In this thesis, we have developed a peer-to-peer based *Network Engine* of the Mammoth game research framework.Based on pSense [ASB08], this network layer is flexible enough such that it can be run on different transport protocols. The idea is that players send their game changes directly to other players that are close to them in the game world without going through a server. As clients move, they detect other players with the help of gossiping. Special sensor nodes suggest them to build connections to new players. Through these message exchanges, each client node creates and updates the list of peers interested in their movements. Since clients constantly move around, this overlay maintenance is highly dynamic. A *Suggestion Engine* is built to perform this overlay maintenance. Experiments are designed to analyse and compare the performance of the network engine when running on top of two different transport protocols: UDP/IP and TCP/IP.

Résumé

Les jeux en ligne massivement multijoueur ont gagné une si grande popularité au cours des dernieres années que leur nombre de consommateurs a explosé. Généralement conçu pour une architecture client-serveur, celui-ci subi une faiblesse au niveau de leur extensibilité, car un goulot d'étranglement se forme normalement du côté du serveur. Afin de régler ce problème, les architectures poste-à-poste ont été adopté et intégré dans les contextes de jeu.

Pour cette thèse, nous avons développé un moteur de réseaux (*Network Engine*) conçu pour une architecture poste-à-poste pour Mammoth, qui est un logiciel intégré pour la recherche des jeux en ligne massivement multi-joueurs. Basé sur pSense [ASB08], notre couche de réseau est assez flexible afin qu'il puisse supporter différents protocoles de transport. Les noeuds clients se découvrent entre eux part l'intermédiaire d'une troisime entité. Cette dernière émet des messages de suggestions afin d'établir des connections entre les noeuds clients. Grâce aux messages communiqués, les noeuds clients créent et fait un mise à jour de leur liste de noeuds homologues intéressés par leurs mouvements. Vu que les clients bougent constamment dans le jeu, cette structure de réseau doit tre très dynamique. Un moteur de suggestion (*Suggestion Engine*) est créé pour faire la maintenance de la structure. Une suite d'expériences ont été développées afin d'analyser et de comparer la performance du moteur de réseau lorsqu'il est exécuté avec deux différents protocoles de transport : UDP/IP et TCP/IP.

Acknowledgments

This work would not have been complete without the support of many. First, I would like to thank my supervisor Bettina Kemme for her constant guidance, support and encouragement throughout the completion of this work.

Finally, I would like to give special thanks to my parents, my sister and the rest of my family and friends for putting up with me and for their constant support and encouragement throughout my studies.

Contents

List of Figures

Chapter 1
Introduction and Contributions

1.1 Motivation

The popularity of Massively Multiplayer Online Games (MMOGs) has increased tremendously. Over the last decade MMOGs consist of a virtual game-world, generally hosted on the Internet, where many players can log-in and possibly interact with all other players in the game world. For fairness purposes, the game must provide the same knowledge about the current game state to every player. To do so, a great amount of data needs to be transmitted over the network.

Traditionally, games are built on a client-server architecture. This offers more control to game companies in terms of managing player accounts, maintaining game state consistency, and detecting and resolving cheating situations. However the client-server architecture does not scale well. A bottleneck generally occurs on the server as more players connect to the game. Most companies deal with this problem by adding more servers and creating server clusters in the system. Although this alleviates the scalability problem, it does not get rid of it. There are costs associated to this practice which include the cost of buying the machines, replacing computer parts, renting a facility to store them, hiring people for their maintenance, other utilities, etc. Companies compensate these costs by asking for a monthly or annual membership fee from their players.

To give smaller game companies a chance to get a piece of this market, numerous approaches have been proposed for utilizing a peer-to-peer structure for MMOGs. The main advantage of a peer-to-peer architecture is its limitless scalability. However, developers must deal with some problems that were inexistent or relatively simple to solve in client-server architectures. Some of these new challenges are player's interest management, game state distribution, and cheat detection and prevention.

Part of the game experience in MMOGs is to play in a virtual world shared among a large number of users. In an ideal environment, each player object will know about every other player object in the game as its decisions might be affected by others' actions. However, this is not a plausible solution for MMOGs due to its poor scalability. As the number of player objects in the game grows larger, the number of messages exchanged over the network and processed at each client node increases greatly. To remedy this, several *interest management* algorithms were introduced to MMOGs. Interest management is the mechanism used to determine which information is relevant to a player object. The most common type of perception in MMOG is bounded by what a player object can see, but is not strictly limited to proximity. For example, if a game object is close to a player object, but they are separated by a wall, the game object becomes irrelevant to that player object.

The Mammoth research group at McGill University has built a massively multiplayer game research framework, named Mammoth. This framework aims to ease the implementation of new game algorithms and provides a means to evaluate them in a real game environment, instead of using simulations. One of the major components found in Mammoth is the network engine. This component implements the core primitives for communication between clients and server. It handles connections and message transmissions using TCP/IP and supports several communication paradigms. However, most of these models are designed for the client-server structure.

Each of the peer-to-peer approaches that have been proposed in the past focuses on some of the game aspects. In the case of pSense [ASB08], the emphasis is put on fast dissemination of player position update messages to interested peers. Being a peer-to-peer approach, each client node must perform the interest management of its player to determine who should receive its updates. This is a difficult task to do on its own. Therefore peer nodes emit suggestion messages to a client node, notifying it about other nodes that might

be interesting to that client node. By doing so, a peer-to-peer structure is formed and maintained dynamically. PSense introduces the concept of near and sensor nodes. Near nodes are nodes that are close to a player object and whose state updates might have a direct impact on the player's actions. Sensor nodes are nodes that are located around a player object in a certain distance and that find other nodes that the player might be interested in.

The task of this master thesis is to build a peer-to-peer network engine capable of listening to suggestions made from peer nodes. This new engine, referred to as Yobol, is an extension module to Mammoth that merges game and network awareness. It is flexible enough to allow executions over different type of transport protocols like TCP/IP and UDP/IP. Based on the pSense algorithm, Yobol contains a *suggestion engine* which is the component responsible of determining a client's interest management and making suggestions to peer nodes such that new connections can be established.

1.2 Contributions

Specific contributions of our work include:

- We provide a design and implementation of a network engine for communication over a peer-to-peer structure in Mammoth. The modular design of our network layer allows for quick and easy swapping of transport protocols, such that Yobol can be run over TCP/IP or UDP/IP.
- We provide a design and implementation of a suggestion engine for maintaining the peer-to-peer overlay according to the pSense algorithm [ASB08].
- We give and discuss experimental results on the usage of TCP/IP and UDP/IP as transport protocol in Yobol for MMOGs.

1.3 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 provides some background information and lists related work on some game concepts and peer-to-peer solutions for

MMOGs. Chapter 4 describes the internal functioning of Yobol and its integration in Mammoth. Chapter 5 gives analysis results of the performance of Yobol executing over different transport protocols (TCP/IP and UDP/IP). Finally Chapter 6 concludes this work and suggests future directions for research.

Chapter 2
Background and Related Works

2.1 Massively Mulitplayer Online Games Overview

Massively Multiplayer Online Games or MMOGs are mostly distinguished by their large number of players (usually in thousands) simultaneously connected to a persistent virtual world. The large scale of the number of players subscribing to these games clearly differentiates them from other network based online games. For example, World of Warcraft is a popular commercial game which currently has over 10 million subscribers, where more than 2.5 million of them are from North America [Ent]. In nearly all multiplayer games, players take control of a game character, also known as an *avatar*. This is the representative of the player in the virtual game world. It is through their avatar that players can see and hear things occurring in the game. Therefore, the game area visible to the player is limited by the abilities of their avatar. Other examples of well known MMOGs are Neocron Evolution [AG], Battleground Europe [Pla], and GuildWars [Are].

The game settings can largely differ depending on the nature of the game. It can take place in a fantasy world with dragons and magicians or a world full of aliens and spaceships. Regardless of the setting, the game world or virtual world usually consists of different types of objects, that can be classified into four main categories: player characters, non-player characters (NPCs), mutable objects, and immutable objects. Player characters or avatars are controlled by the players. The state of a player character usually contains its

current position in the game world, running directions, abilities, health or its possessions. The avatar's possessions generally consist of items collected by the avatar while moving in the virtual world. Aside from player characters, there are also non-player characters like monsters or enemies. These are very similar to player characters, but differ by the fact that they are controlled by an AI algorithm. Mutable objects, like food and weapons, are objects with properties that can be modified during the game. For example, a door that can be opened or closed is a mutable landscape item. Its state would be determined by a position and a status such as "door is closed". Lastly, immutable objects or static objects do not have any properties that can be changed during the game. Their states can be represented as a 2- or 3-dimensional vector. Some immutable landscape item examples are doors, windows, rivers and trees.

The collective states of all objects in the game at a given time constitute the *game state*. Objects' states are modified when an action is performed by or on them. There are three main actions that can be performed. The first one is the change of position. When an avatar or a NPC moves around, its position is modified along with the game state. The second is the interaction between players which can alter the state of one or both parties. For example, if two players fight against each other, the health of both players will decrease. The third type of action is the interaction between a player and an object in which the state of both entities gets changed. For example, if a player drinks water from a bottle, the player is not thirsty anymore and the bottle contains less water.

2.2 Interest Management

In order to provide a shared sense of space among players, each player must maintain a copy of the (relevant) game state on his/her computer. When one player performs an action that affects the world, the game state of all other players affected by that action must be updated. The simplest approach is for each player to maintain a full copy of the game state and all players broadcast updates to all other players. The problem with this approach is that it does not scale well: as the number of player increases, the number of messages sent over the network and to be processed by each client greatly increases.

One of the most effective strategies to address this problem is to send to a player's computer only the messages that are relevant to its avatar (e.g. only the update message of objects it can see, or that are near). The world space of MMOGs contains a lot of information and a single player needs only to know about a subset of that information. *Interest Management* (IM) is the process of determining which information is relevant to each player [Mor96].

The information relevant to a player usually corresponds to the perception of its avatar. In the interest management scheme, this is often based on proximity, modeled as a sphere around the avatar. However, the most common type of perception in MMOG is what an avatar can see, which does not always correspond to proximity. In particular, game worlds usually contain static obstacles that occlude regions of the game space. For example, if an object is close to an avatar but is behind a wall, it becomes irrelevant to that player.

2.2.1 Euclidean Distance Algorithm

In the Euclidean distance algorithm, the area-of-interest is a circle around the position of the player. The radius or *vision range* determines the maximum distance a player can see. If the distance between an object and a player is smaller than the radius of the area-of-interest, the object is declared to be in the player's vision range. Then the player subscribes to all objects located in its vision range.

2.2.2 Tile Algorithm

In tile algorithms, the world space is divided in tiles or *zones* and the player subscribes to all objects in the tiles that intersect its area-of-interest. Tiles can be formed of various shape such as square, rectangle or hexagonal. These types of approaches try to leverage the occlusion created by obstacles in the world. An example of a tile algorithm is the square tile algorithm. It is a zone-based interest management such that the game world is divided into equal-sized squares. The radius of the area-of-interest of the player determines the number of the squares. For example, at any given time, an avatar subscribes to at most 9 zones, the one it currently resides in and the 8 (or less) neighbor tiles around it, as depicted in Figure

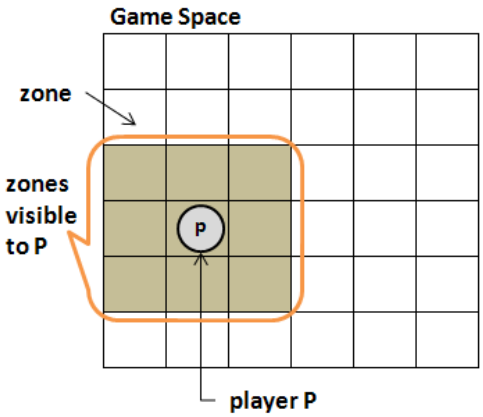


Figure 2.1: Tile Algorithm Example

2.1 When a player performs an action, the action is broadcast to all players subscribed to the square in which the action has taken place [JSBV06].

2.3 Existing Architectures

There are several ways to implement the network structure of MMOGs. Three main ones are client-server, pure peer-to-peer, and hybrid peer-to-peer. Each approach has its advantages and disadvantages, ranging from scalability to complexity of implementation. In this section, an overview of the three above mentioned network architectures is described.

2.3.1 Client-Server

The most commonly used approach in MMOGs is the client-server architecture. The server acts as a central component responsible of maintaining the game state, meanwhile the client hosts the user's avatar and handles the avatar's state updates. To start playing, clients must connect to the server to retrieve the latest game state and store it locally. Afterward, all actions performed by clients are converted to state updates, which are wrapped in messages, and are sent to the server. Upon receipt of clients' update messages, the server deserializes them, processes them to generate a response, serializes the response and multicasts it to all clients that have an interest in this update according to the interest management. Clients

receive the response message and use it to update their game state.

This model is chosen in most cases for its ease of implementation and control, such as detecting and preventing cheating amongst players or adding object persistence in order to handle system crash. Since the server is the main cause of the system bottleneck, more than one server can be used to support a larger number of players. They can also be arranged in clusters. Usually, in order to avoid coordination among servers, each server runs its own instance of the game with a limited number of clients. Generally, for scalability reasons, game companies' servers are located in huge server-farms. In January 2008, World of Warcraft recorded over 10 million users [Ent]. Being one of the largest game company, it can afford purchasing more servers to support the increasing user's demands. However, smaller or start up game companies cannot follow this strategy as they will not be able to afford such setup. For these smaller game companies, using the client-server model is not that suitable since the cost of servers will increase very fast which poses a limit to the number of servers they can have. The maximum number of participating players is limited by the maximum workload of these servers. Another point is that if game companies want to look into a different payment method, then they cannot afford the large server farms. Hence, other architectures need to be explored.

2.3.2 Peer-to-Peer

An alternative model is the peer-to-peer architecture where participating nodes or *peers* provide a portion of their resources, like processing power or network bandwidth, to other participants in the network. In contrast to the client-server model where the server processes and provides information and clients consume it, peers are both information suppliers and consumers. Peer-to-peer networks are typically formed dynamically by ad-hoc additions of nodes. Furthermore, the removal of nodes should have no significant impact on the network. They are highly scalable because the computations, resources and communication overheads are all shared amongst network participants. Their scalability and their capabilities to build and maintain peer-to-peer overlays themselves make them favorable for designing MMOGs network structures.

In a peer-to-peer network, all peers are equal and simultaneously function as both

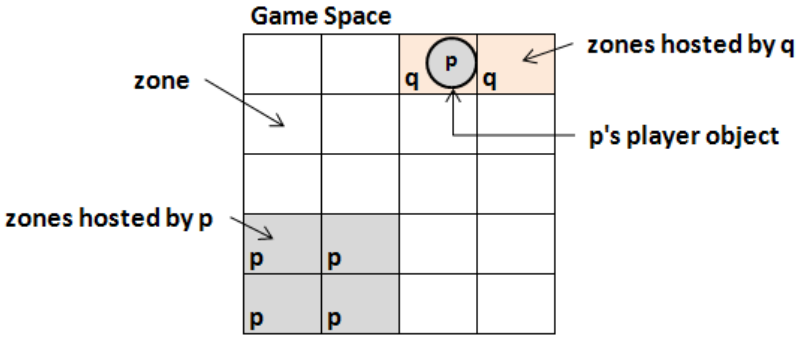


Figure 2.2: Game State Partitioning - Zones

"clients" and "servers" to other nodes on the network, such that no server is needed. In a very common setup, the game world is partitioned into multiple *zones*. In addition to hosting the user's avatar and handling its state updates, peers can be assigned one or several zones and will become their host. This means that such peer will act as a "client" for its user and as a "server" for all objects residing in its zones.

To start playing, clients must connect to a peer node in the network to retrieve the latest game state and store it locally. Then some zones might be assigned to them. Afterward, all actions performed by clients are converted to state updates, wrapped in messages, and sent to the appropriate zone host. Upon receipt of update messages from peers, the host node verifies that the update message concerns a zone it is managing before it processes the update. Then it prepares a response message and multicasts it to all interested clients. When peers receive a response message, they apply the informed changes to their local game state. Figure 2.2 shows the division of the game state into small rectangular zones where peer p is the host of 4 zones and peer q manages 2. The avatar of node p is located in one of the zones managed by q. When the avatar moves, p sends an update message to the host node q, who in turn processes the update, generates a response and multicast it to all peers interested in the movement.

The challenge of this game world split is that if an action is interesting for players that are outside the zones a peer handles or beyond the region which the players reside in, the neighboring zones are difficult to detect. Thus, in most implementation, player's vision range does not go beyond the zone it is residing in.

Pure Peer-to-Peer

An example of pure peer-to-peer system is Solipsis [JK03]. It is a mathematical model for a massively multi-participant shared virtual world. The peer-to-peer network of Solipsis is modeled by a graph that consists of a set of nodes and a set of connections between those nodes. Each node has a unique ID and is responsible of its own state. They are able to sense part of the virtual world, which is inhabited by other entities. Their neighbors should recognize by their own initiative that the status of the node has changed. Therefore, each node has to know all the entities that are in a certain region around it. Moreover, the system must ensure that no node will be disconnected from all others. Each node should somehow be connected via other entities to every single peer in the system. However, up to date no real game implementation of Solipsis exists, making this approach more theoretical than practical since no overhead or performance evaluations have yet been published. This makes it difficult to judge its suitability for MMOGs.

This type of architecture is very scalable as no server is needed. Nevertheless, there are many unsolved problems like cheating, player authentification, and game persistence, making it not readily suitable for MMOGs.

Hybrid Peer-to-Peer

The idea behind hybrid peer-to-peer architectures is the usage of peer-to-peer protocols while still maintaining servers. This can be found in games like GuildWars [Are] and Neocron Evolution [AG]. These servers do not have so many tasks to fulfill other than supporting the peer-to-peer structure of the nodes, such as authorization and authentification making sure that only credible users can access the game, storing players' persistent states like account-information or the avatar's abilities and possessions. The game state is taken care of by the players themselves, and not the server as discussed in the previous section. However, by giving more control to the players, there is a greater possibility for cheating.

Hybrid peer-to-peer offers the advantages of both client-server and peer-to-peer systems. It is more scalable than the traditional client-server architecture and less costly to develop as no such expensive server-clusters are needed to handle the number of players. On the other hand, players will have to participate in keeping the game alive by providing

some upload resources. Moreover, since the hybrid approach has a server in place, cheating is relatively easy to control. Every action that takes place in the game will be transferred to the server. It can control and verify the changes, such that if any irregularity arises, the centralized server can decide if this was due to cheating or not. This solves the biggest problem of pure peer-to-peer architectures. The approach introduced in this thesis follows the hybrid peer-to-peer structure where peer nodes only handle information related to position updates. The server is responsible for authentification, player distribution, persistence, and keeping the game state of mutable landscape items consistent.

2.4 Other Peer-to-Peer Structures in MMOGs

Many different approaches for using peer-to-peer algorithms for MMOGs have been proposed. In some approaches, messages are multicast using a distributed hash table (DHT) or a dissemination tree. Others are built on group communication systems or gossip protocols.

In some works, the game world is divided into small sub-spaces or zones which will be distributed over all participant nodes in a peer-to-peer network. SimMud [BK04] follows this approach by using a Pastry DHT [RD01] to map the client nodes and the game objects (zones, players, pick-able flower, etc.) to an identifier randomly chosen from a uniformily distributed 128-bit name space. The game sub-spaces and objects are assigned to client nodes whose identifier is closest to the object's identifier, and thus the client becomes their master node. The DHT is used to lookup where a particular object is located. On top of the DHT, SimMud uses Scribe [MCR02] to multicast the game state. For each existing sub-space, a multicast group is dynamically formed between players located in the same sub-space. This is made possible by the self-organizing properties of peer-to-peer networks. In order to see what is happening in the game, a client node has to join to as many multicast groups as necessary, determined by their interest area. As players move around, their multicast groups will change accordingly. Among all client nodes in any multicast group, one is chosen as the group *coordinator*. The role of the coordinator is to be the root of the multicast group as well as to provide newly joined clients in the group with the most updated state of the sub-space.

Some improvements of SimMum were later proposed in Mopar [YV05] where a few optimizations were done to reduce latency. This was achieved by making newly joined clients in the group retrieve the current game state from the coordinator's local cache instead of getting if from the DHT. This consequently reduces the usage of the DHT to backup data storage. Since accessing data directly from the coordinator is faster than from the DHT, the amount of time elapsed before a client receives the game state is greatly reduced. Similarly, in Mercury [ARBS04], a *rich subscription* language was developed to allow players to express their interests with more flexibility. It also provides an efficient routing mechanism to multicast publisher's updates to their subscribers. Nevertheless, all these rely on splitting the game world into smaller zones. Zone-based approaches comport many disadvantages, such as expensive hand-shakes when a player moves from one zone to another. Also if a coordinator leaves the game, an expensive reconfiguration is required. Due to the dynamic nature of games, the number of players can constantly vary, making load-balancing difficult to perform.

A dissemination tree [MCR02, LP96] is often used to multicast game states messages across all nodes in many peer-to-peer systems. When tree-based multicast protocols were developed, they were mainly aimed for applications with a large receiver base. This is not the case in MMOGs because generally a message is only interesting to a few players. Some argue for using group communication systems (GCS) [GCV01] instead, as these offer primitives to multicast messages to a group of sites. However, determining when a player has to join which group is complex. Also, the actions of joining or leaving a group are generally expensive operations.

Gossiping protocols [PTEK03, KPBM99] can also be used in MMOGs to multicast direct messages or recover lost messages. They aim at propagating messages of all nodes to every other node in the system, thus achieving high reliability. However, this creates a lot of redundancy in the system, which is counter-productive in a large peer-to-peer game environment because peer nodes are likely to be overloaded with the large amount of messages to be processed. Additionally, in a peer-to-peer MMOG setting, each message should only reach a small subset of players since only those interested in the update should receive it. Therefore using a localized multicast mechanism would be more suitable for this context.

2.5 pSense Algorithm

pSense is a localized multicast for fast dissemination of player position updates in a dynamic game setting [ASB08]. It exploits a peer-to-peer structure for better scalability, as it is one of the main problems with traditional client-server architecture. This section summarizes information presented in [ASB08]. A general idea of the pSense algorithm is given, followed by a detailed description of the overlay maintenance, and the joining and leaving of a node in the network.

For ease of understanding, the distance between two nodes refers to the distance between two players hosted by the nodes in the game world. The area of interest defined by the interest management is called the *vision range*. The interest management described in pSense is the Euclidean distance algorithm. Since every node in the game hosts a player, the player hosted in node A is referred to as player A. A random node B is a *near node* of A if the player character hosted by node B resides in the vision range of player A. The words player and node will be used interchangeably in the following.

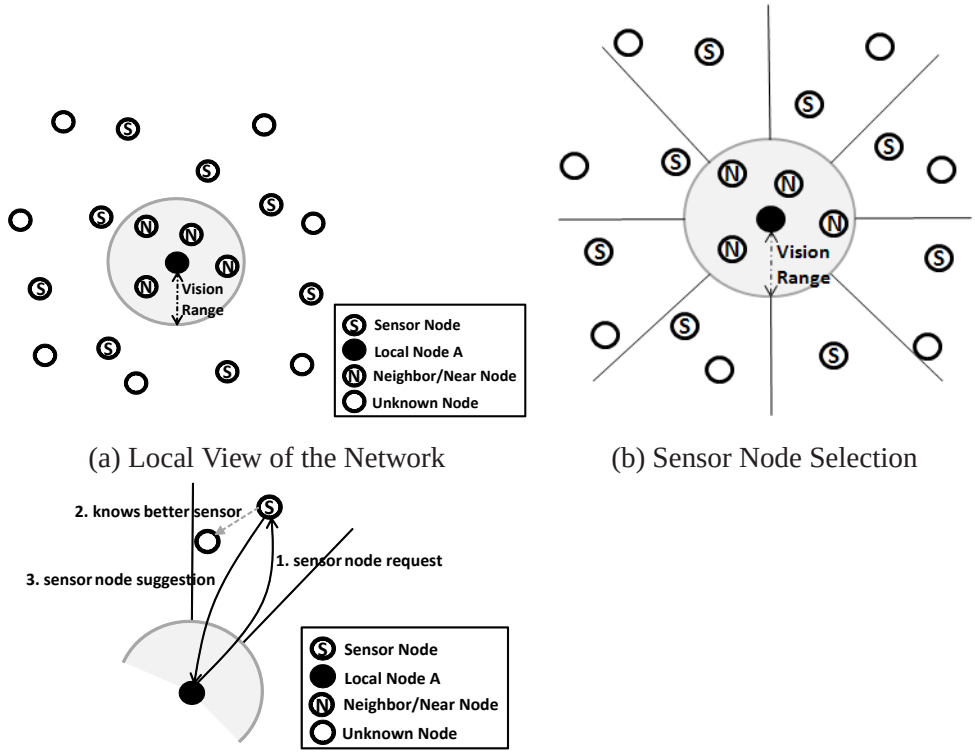
pSense functionality can be summarized in two main tasks. The first task consists of sending the position updates of a player A to its near nodes when it moves. Sending the update message to every other player in the game is not very scalable. For example, assume there are 200 active players in a game, and each player moves to a random position every 30 seconds. This amounts up to 80,000 position update messages sent every minute. As the number of players increases, the network runs a greater risk of getting overloaded. With 2000 clients, there will be close to 8,000,000 messages in the network per minute just for position updates. For scalability reason, position update messages will only be sent to near nodes located in the vision range. It is assumed that the relation between a node A and its near nodes are symmetric. This means that if A is interested in node B, then B is also interested in player A.

Position update messages are not to be strictly sent to immediate near nodes. Doing so will greatly risk the creation of game network partitions. Network partitions occur when agglomerations of players are formed in the game, where each are situated far apart. A disconnection between those groups can occur, such that no player from one group is aware of what is happening in the other group. Players then lose valuable game information. Once

network partitions exist, they cannot be fixed since no superpeer node with global knowledge exists in pSense. Thus, the second task of pSense is to keep the network connected. This is achieved by having each node maintain a list of *extended nodes* known as *sensor nodes* in pSense. Sensor nodes of a node A are nodes just a bit outside of the vision range of A. They stick out in every direction like antennas. In Figure 2.3(a) the black dot represents a node A, referred to as *local node*. The circle around it is its vision range. All nodes residing in the vision range are maked with *N* indicating immediate *near nodes*. Figure 2.3(b) shows the space division around the local node A where 8 pie sections are formed. Each pie section has an identifier and each sensor node around node A is assigned to a specific section. There is only a single extended peer node per section. Routine checks will be performed to look for the best sensor node candidate for a given section. pSense only considers a 2D space using 8 sectors to divide the space around the local node, such that extensions might be needed for 3D. Moreover, other types of division can also be implemented, for example using triangles instead of pie shape sections. Finding an optimal division configuration is however out of scope of this thesis.

Sensor nodes have two main purposes. First, they prevent the creation of game network partitions by broadening a nodes knowledge of the game. A node A stays informed of what is happening in its vicinity by the near nodes, and gets some knowledge of what is happening outside its vision range from its sensor nodes. Second, they help a node A to quickly detect players that enter or leave its vision range. For example, assume that a player P enters the vision range of node A and A and P do not know each other. Since they do not have any knowledge of each other, it is difficult for either one to initiate the contact. Assume that one of the sensor nodes of A knows node P and has been getting updates about P for some time. Once the sensor node sees that P has entered the vision range of A, it notifies node A about the newcomer with a message containing information about P.

In other words, pSense must restrict sending position updates to near players, while avoiding creating network partitions. This is achieved by making every node keep two distinct lists: a list of near nodes and a list of sensor nodes. The former contains only the near nodes that are within the local node vision range which need the position updates very fast. The later contains nodes that are just a bit outside the vision range, sticking out in every direction. The sensor nodes of a node A should be distributed as evenly as



(a) Local View of the Network

(b) Sensor Node Selection

(c) Sensor Node Maintenance

Figure 2.3: pSense Algorithm figures adjusted from [ASB08]

possible around A for a better chance of keeping connections to the rest of the network, thus avoiding network partitions, and efficiently detecting new approaching nodes.

2.5.1 Sensor Node Selection and Maintenance

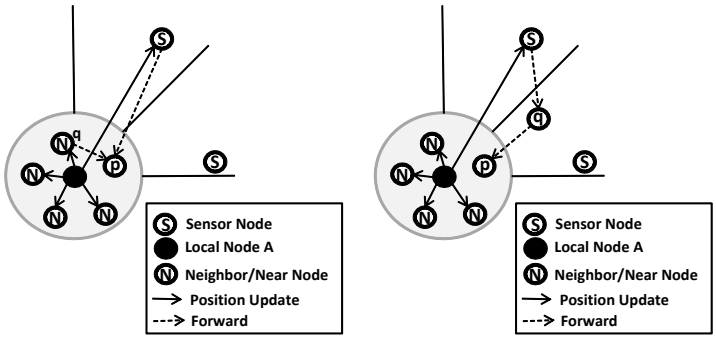
Sensor nodes of a node A reside a bit outside of the vision range of A. They should be distributed as evenly as possible around A. Figure 2.3(b) shows an example of how sensor nodes should be distributed. Since node A does not have much knowledge outside its vision range, it can hardly find the best sensor candidates on its own. Therefore, it periodically sends *sensor request* messages to the sensor nodes asking them if they know a better candidate for this section. The sensor request message contains the position of A and the section identifier. The sensor node checks if there is a node better suited than itself (or could be itself) and sends a *sensor suggestion* message back to A. The sensor suggestion message

contains the identifier of the suggested candidate and the section identifier. Node A then replaces the old sensor node with the new one. This is depicted in Figure 2.3(c).

2.5.2 Message Forwarding

According to pSense, whenever a player A moves in the game its new position is sent to its near nodes and sensor nodes. However, every machine has a limited amount of upload bandwidth capacity. Therefore, it may not be possible to send the update to every near and sensor node. When the number of nodes in these two sets exceeds the upload bandwidth capacity, the update is only sent to a random subset. These nodes receive the new position directly from player A, using one hop. Upon reception of the update, these nodes check in their own list of near and sensor nodes, looking for nodes that reside in player A's vision range (the message originator) and which have not received the update yet. In Figure 2.4(a), we assume that node P has either recently entered the vision range of A, or it s not part of the arbitrary subset of near nodes chosen to receive the position update. Near node Q and sensor node S_1 are closer to P, therefore we assume that they already know about P. When Q and S_1 receive a position update from player A, they find out that P is in the vision range of A and has not received the position update yet. Either one of them or both will then forward that update message to node P. P learns about A and puts it in its near node list. In the next round, P sends its position update to A, which then puts P in its near node list, if it is not yet there. In some cases, the sensor node cannot find a new node located in A's vision range. It then looks for nodes that are closer to node A. These will in turn forward the message to other nodes that are closer to node A. This step is repeated until the message is no longer deemed interesting. In Figure 2.4(b), sensor node S_1 does not know node P, but sees that node Q is getting closer to node A and may be interested in the update. S_1 forwards the update message to Q. Having knowledge of node P, Q sees that P is in the vision range of A and has not received the message yet. Then Q forwards the position update to P, thus enabling node P and A to discover each other.

Message forwarding plays a crucial part in creating and maintaining the overlay. It allows node A to detect new near nodes. However, position update message are not to be forwarded indefinitely. This will create unnecessary duplicate messages in the network.



(a) Localized Multicast (b) Indirect Forwarding

Figure 2.4: Message Forwarding adjusted from [ASB08]

Players are not interested in receiving an old position update when a fresher one has already been received. Aside from limiting the life span of position update messages, detecting and filtering out duplicate messages is necessary in order to maintain a lower load and bandwidth consumption.

2.5.3 Overlay Maintenance

This section details the steps taken to process received messages, to determine and to send outgoing messages.

Filter Receive Messages

An arbitrary node A can receive a new message at any time. A received message could be a position update message or a sensor request or suggestion message. All messages are tagged with a unique node identifier and a sequence number.

When a new message X is received, its hash is compared to a list of seen hashes. All duplicates get discarded. This ensures that no duplicate messages will be processed. If message X is a position update, then it is compared to the last position update message Z received by the same sender. If the sequence number of X is smaller than the sequence number of Z, then message X is discarded as it is not the most recent position update about that player. All expired/old messages are discarded at this point. The remaining messages are put in an incoming message queue, to be processed in the next step.

Overlay Maintenance and Multicast

Periodically, node A performs the following actions in order to maintain the multicast structure.

1. **Update Near Node Lists**:

   Node A updates its list of near and sensor nodes by first checking all position updates and sensor suggestion messages contained in the incoming message queue. All nodes residing in the vision range are put into the near node list. Those suitable to be sensor nodes are selected and put into the sensor node list (see Section 2.5.1 for details). The rest are discarded.

2. **Determine Outgoing Messages**:

   Since message sending only occurs once per round, all messages to be sent are stored in an outgoing message queue. Position update messages have a life-time limit. This is measured in the number of hops travelled (one plus the number of forwards). If the life-time limit of a message is reached, it does not get forwarded anymore as the information is no longer considered *fresh*.

   First, a position update message is created with the current position of player A and the recipient list. This list contains the identifiers of all nodes in its near and sensor list. This will help reducing the number of duplicate messages in the network. Second, a sensor request message is sent to each sensor node in order to get the best candidates. Finally, all messages in the incoming message queue are processed. If it is a sensor request message, then a sensor suggestion message is created (see Section 2.5.1) and put in the outgoing message queue, to be sent back to the message originator. If it is a position update that has not reached its life-time limit yet, then node A checks its near and sensor node list for near nodes of A that have not received the update yet. If no near nodes are found, it checks for nodes that are closest to the originator. This is done by sensor nodes after receiving a position update from node A. The recipient list of the update message is modified to include these new receiver nodes. The update message is then put in the outgoing message queue.

3. **Send Messages**:

   Before messages contained in the outgoing queue are sent, a final verification is performed. If the outgoing message queue exceeds the upload bandwidth of node A, some randomly chosen near nodes in the recipient list are deleted from the list until it is small enough to be sent. The update message is then sent to each node in the recipient list. For example, player A sends its new position to its peers B-Z. The recipient list in the update message will contain nodes B-Z. However its bandwith limitation prevents it from sending to all its peers at one time, such that some randomly chosen nodes are deleted from the list. The recipient list in the update message is modified such that it only contains only node B, D, G, and H. They will be they only nodes to receive the position update message directly from A. Although it is acceptable to delete some update messages in order to meet bandwidth capacity because they can be recovered through message fowarding, sensor suggestion messages and sensor request messages never get deleted. Once all messages are sent, both incoming and outgoing queues get cleared.

2.5.4 Joining and Leaving the Network

To join, a new node only needs to know a random node already existent in the network. This existing node is referred to as the *old node*. If the old node is in the vision range of the new node, position update messages are sent directly between them. If the old node is not in its vision range, the new node sends a sensor node request. The old node checks in the list of known nodes and suggests a better node than itself. This is repeated until the new node finds the best sensor node candidates. Meanwhile, these sensor peer nodes also receive the position updates of the new node. These update messages are then forwarded to nodes that are closer to the new node. It will eventually reach a node that is within the vision range of the new node, which will enable the latter to build its own near node list. When a node leaves the network, no special operation is needed. If a peer node is lost, the others simply stop sending update messages to that node. If a sense node is lost, a new one is chosen as described in Section 2.5.1.

Bootstrapping

pSense does not state how to start a game, but suggest a central server which players can connect to and get all necessary information about one player already connected to the network. An assumption is made that after the first node has joined the network, the system will automatically bootstrap itself. Other concepts like player authentication, game state distribution, and object replications are not covered in pSense.

2.6 Mammoth

Mammoth is a massively multiplayer game research framework implemented in Java. Its goal is to provide an implementation platform for academic research related to multiplayer and massively multiplayer online games in the fields of distributed systems, fault tolerance, databases, networking, concurrency, artificial intelligence, modeling and simulations, aspect orientation and content generation [JK09, Zin08].

Like other multiplayer games, Mammoth requires players to log in and take control of an avatar. During the game, the avatar can move around the virtual world and interact with the environment, like picking or dropping items like flowers, and communicating with other players through a chat box. When picking items, they are put in the avatar's inventory, which may have a limited capacity. Other than player characters, there are also immutable landscape items like trees, walls, fountains, and cars. They act as obstacles preventing the avatar from moving in a straight line. Currently, there is no specific goal to achieve in Mammoth.

2.6.1 Server Architecture and Object Replication

Game objects in Mammoth are distributed according to the *distributed object model* methodology created by Quazal Inc. In this approach, the entire game state can be described as a collection of objects where each object has a particular state. These objects are distributed across all client machines participating in the game. In MMOGs, it is essential that the state change of an object in a machine will be visible to other machines. Objects must then

be *duplicated* over the network to other machines in order for them to see these modifications. An object can either be a *master object* or a *duplica object*. As the name suggests it, the master of an object is the controlling instance who performs all changes to the object's state. Duplicas are copies of the master object which are sent to other machines. They frequently get updated in order to keep their state consistent with the state of the master object. To be able to control an object, such as a game character, clients will get a copy of their player object, known as *master duplica*. In other words, in a client-server architecture, all master objects reside on the server. Clients get object duplicas when they subscribe for updates and get a master duplica of their game character which enables them to control the avatar.

Being a client-server based MMOG, the Mammoth server holds the master objects of all player and other mutable objects in the game. When starting a game, clients connect to the server to retrieve their avatar information and its master duplica. After receiving the master duplica, clients can start moving their avatar in the game world. The interest management of the server determines what is visible for each client. These can only see changes after they have received a duplica object from the server. Whenever the avatar moves, the master duplica on the client node redirects the operation to the master object on the server by a remote call. It is asking for the permission to change from the master object. If the master grants the change, it performs the update on itself first and generates a response message. This response message is then multicasted to all clients holding the corresponding object duplica (i.e. subscribed clients). For consistency reasons, actions like player movements are always performed at the server holding the master object first. Figure 2.5 illustrate the entire process, client 1 joins the game and is assigned player Bob. It gains control of the game character named Bob by holding the master duplica. Meanwhile, a client 2 might be interested in seeing Bob's state updates and therefore has a duplica of Bob. How it receives such duplica will be described in the next section. Whenever the avatar Bob makes a movement, its new position is sent to the server who processes it, creates a response message and multicasts it to all subscribers of object Bob about the update.

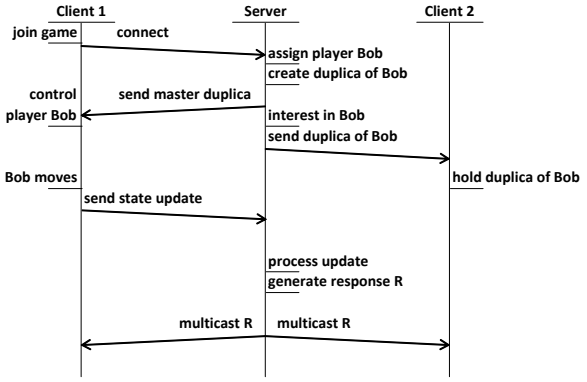


Figure 2.5: Player State Update Dissemination - Client-Server

Object Migration in Mammoth

Mammoth currently only supports one model of object migration called *Burst Migration*. It is designed for distributing objects onto several machines. This could be used in a server cluster system where, if a server machine becomes overloaded, it can migrate some of its master objects onto other servers to alleviate its load. It could also be applied in a zone approach (see Section 2.2.2) where the host of a given zone also holds the master objects of all objects located in the assigned space. This migration model is called *burst* migration because the sever sends the master object to the target destination without prior notice or proper handshake procedure to verify if the receiver end is ready for the migration to take place or not.

2.6.2 Publish/Subscribe Basics

Publish/subscribe systems are designed for situations where a large number of subscribers, with diverse interests, have to be notified about an event or a publication. Also, whenever asynchronous communication is necessary, publish/subscribe can be used, such that the publisher does not wait for an answer from the subscriber before continuing. Such systems are highly scalable because clients do not need the global knowledge of the network. Subscribers do not know the publishers, and publishers do not know the consumers. Publishers

are responsible of submitting data as publications or notifications, whereas subscribers subscribe to publications. Both publishers and subscribers are clients. Moreover, a client can be both a publisher and a subscriber at the same time. There are many ways to implement the publish/subscribe scheme, often there is an event server and all communication is through this server.

The most common publish data model is the topic-based model. With topic-based publish/subscribe, "subscribing to a topic T can be viewed as becoming of a member of a group T, and publishing an event on topic T translates accordingly into broadcasting that event among the members of T" [EFGK03].

2.6.3 Interest Management in Mammoth

Interest management is currently done in Mammoth at the server. The server also has all master objects but the design allows master objects to reside at any nodes. For example, the master copy of a player could reside on the node of the player. When the interest management module determines that a player A should know about player B, the client node that controls player A should subscribe to B. This means that the client node of player A needs to receive a duplica of player B. And then it needs to receive all changes like position updates that occur on B.

Mammoth does interest management, i.e. determining who should see whom with the help of a duplication space. In the current version of Mammoth, only the server has a duplication space which contains the master objects of all mutable objects and players. Both mutable objects and players can be publishers as their state change. Only players that are currently logged into the game are potential subscribers as their client modules need to know about the objects and players of their player's vision range. Each publisher A is associated with a topic or channel X, and all its actions are published on this channel. When a player node subscribes to this channel, its client first receives a duplica of A and later all updates published on the channel. How this message dissemination is done is left to the network engine.

Matching Policy

The matching function checks whether a mutable game object/player might be of interest for an active player. Various matching functions can be used as they are specific for the game semantics. They take vision range characteristics into account. Mammoth executes periodically the matching function on every possible object/player-vs-active player publisher/subscriber pair. If it returns true for the first time for a given pair, the active player becomes a subscriber to this object player. The server creates a duplica object of the publisher and sends it to the client node of the subscriber. All eventual state update made to the publisher object are propagated to all subscribed clients. Thus, clients are kept informed about the current players or other game objects located in their interest area and receive their state updates.

Refresh Interval

The refresh interval refers to the amount of time elapse between each execution of the matching policy on the server node. The chosen value plays a major role in determining how fast a player can discover objects of interest in the game. A too large value will lead to late object discovery and a too small value will consume server resources unnecessarily.

2.6.4 Mammoth Components

Figure 2.6 gives an overview of the Mammoth software design. It follows a very modular approach where each *component* implements a major functionality of the system. *Submodules* are used to encapsulate specific concerns or features in the components. Components provide well-defined interfaces to facilitate the interaction between them. Thus, the underlying implementation of each component becomes transparent to others. This abstraction allows the developer to create and experiment with various algorithms pertaining to their interests without having to modify the entire system. For example, if we want to implement a new communication protocol, then only the *network engine* component is modified. No changes will occur in the *client* component. Some major components in Mammoth are:

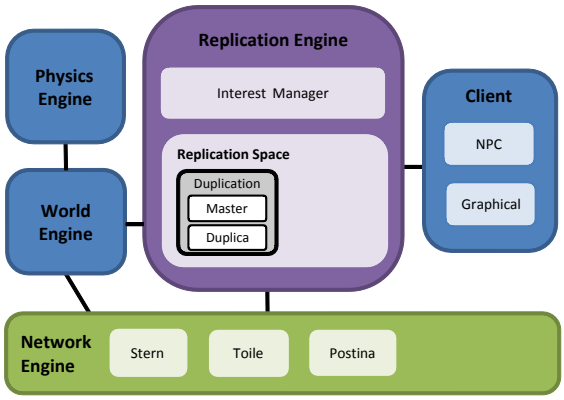


Figure 2.6: Components in Mammoth

- **Network Engine:** The network engine implements the core primitives for communication between clients and server. Currently, connections and message transmissions are made through TCP (see Section 2.7). Game state updates are being transferred as serialized messages. The network engine is the most important component for this project. It is not specific to the game. Several network layers have been implemented in Mammoth such as:

    1. ***Stern*:** In the Stern network engine, a star topology is formed where all clients are connected to a central hub. The hub handles and redirects all network traffic as well as manages the publish/subscribe functionalities. When a player subscribes to a topic X, the hub keeps track of that. When a message is published to a topic X, it is sent to the hub which forwards it to all subscribers.
    2. ***Toile*:** The Toile network engine creates a fully connected network where all clients are connected to all others. Clients can join the network by connecting to a *RendezVous* node, responsible of managing the arrival of new members. The RendezVous node returns the IP contacts (address and port number) of all clients currently in the network. The new client uses this information to connect to all other nodes in the network. Clients manage their own publications and subscriptions locally. When a player subscribes to a topic X, the client node publishing on topic X keeps track of that. When a message is published to a

topic X, it is directly sent to all subscribers.

3. ***Postina*:** Is a self-organizing peer-to-peer network engine using Pastry and Scribe [RD01, MCR02]. It provides publish/subscribe functionalities where clients can publish information to a topic, and all those subscribed to this topic will be informed about the publication. Therefore, to receive a publication, clients must first issue a subscription to the corresponding topic. More details about the Postina network engine is found in [Zin08].

- **Client:** The client module contains the implementation of a graphical or a non-person character (NPC) client. When using a graphical client, a game map is rendered in a graphical display using the OpenGL graphics library. Users click on the screen to set the destination and make their character move. An NPC client is controlled by an AI algorithm. Users may choose to render a graphical display showing the NPC client moving on its own without the need of user interaction.

- **Physics Engine:** The purpose of the physics engine is to implement interactions between two objects following the law of physics. It is currently only responsible for detecting collisions between two objects.

- **World Engine:** The world engine component stores all game objects in several hashtables in order to provide easy and fast access.

- **Replication Engine:** The replication engine is responsible for distributing and updating the state of game objects across clients according to an interest management policy. It has one or more *replication spaces*, each representing a different interest management domain with distinct interest management policy. The replication space controls the propagation of update events for a set of objects in the game state. As soon as an object/player is in the vision range of a player which is determined by the interest management and the replication engine, the client receives a duplica of the object and subscribes to the changes.

2.6.5 Communication Strategy

Mammoth introduces another layer of abstraction to enable clients to keep track of their subscriptions list and to efficiently communicate their changes. This layer is called the communication strategy[1]. Its purpose is to abstract the network communication from the rest of the interest management and object replication logic. This creates better code reuse because the same code can be used on both the server and the client side. Some of the methods found in this communication strategy are:

- **publish (topic):** is used to send a state update to all clients subcribed to the specified topic.
- **replicate (topic):** is used to send a duplica object to a client node.
- **subscribe (topic):** registers the client to a topic such that it will receive the state updates.
- **unsubscribe (topic):** removes the client from a topic such that it will stop receiving their state updates.
- **sendToMaster (topic):** sends a message directly to the node hosting the master object.
- **sendToTarget (topic):** sends a message directly to the specified client node.

2.6.6 Other Services

There are some operations that are orthogonal to the implementation of a game. Example operations are the assignment of an avatar to a user also known as player distribution, authenticating a user or instant messaging. In Mammoth, these are called *services* and reside in a service server. When clients join a game, they first contact with the service server to have their credentials verified. If they are deemed valid, they continue with their communication with the game server to retrieve the states of the game world and their game

[1]The actual class in Mammoth framework is called *Replication Strategy*.

character. However, if the service server cannot validate a client within a certain amount of time, a time-out exception is thrown and the client program is terminated. Otherwise, if the client is invalid, then an authentification error is issued.

2.7 Transport Protocols: TCP, UDP

The User Datagram Protocol (UDP) is a connectionless protocol which does not perform any implicit hand-shaking dialogues. It is only a best-effort protocol that offers no guarantee for reliability, packet ordering, or data integrity. Thus, UDP provides an unreliable service in which datagrams may arrive out of order, go missing without notice or may be duplicated (the same datagram is sent more than once). UDP assumes that error checking and correction is either not necessary or will be performed at the application level. This avoids the processing overhead at the network interface level. The small overhead of UDP makes it appropriate for highly interactive games where the speed of packets delivery is key, like first-person shooter games and car racing.

The Transmission Control Protocol (TCP) is a connection-oriented protocol where a connection between two machines must be established before data can be sent between them. TCP provides a reliable service and guarantees packet ordering. This means that all packets sent are guaranteed to arrive at the destination in the same order that they were sent. Additionally, it uses an end-to-end flow control protocol to avoid having the sender send data too fast for the TCP receiver to reliably receive and process it. This is useful in heterogeneous environment where machines of diverse network speeds communicate. TCP has the advantage of being simpler to use than UDP, but it generates a noticeable amount of overheads.

In the current implementation of Mammoth, communication between clients and server is made over reliable TCP/IP connections. In this thesis, it is in our interest to evaluate the performance of TCP/IP versus UDP/IP in a peer-to-peer context. Although TCP guarantees message delivery, it generates a connection overhead which grows as the number of connections maintained by a peer node increases. UDP is much lighter since no initial connection setup is required, but being an unreliable transport protocol, excessive message

lost can be problematic in a game context.

2.8 MINA

MINA is a framework, made available by Apache, designed to ease the development of high performance and high scalability network applications, by offering an abstract, event-driven, asynchronous API over various transports protocols like TCP/IP and UDP/IP via Java NIO [min]. MINA is the acronym for *a Multi-purpose Infrastructure for Network Applications*.

By using MINA, the effort needed to test the performance of a system using different transport protocols, like comparing TCP with UDP, is greatly reduced. MINA's structure consists of four fundamental components: *IoService*, *IoSession*, *IoFilter*, and *IoHandler*. The IoService provides supports for input and output operations. There are two distinct IoServices available: IoAccceptor and IoConnector. The IoAcceptor acts as a server by waiting for incoming connections, while the IoConnector acts as a client by establishing a connection to the server. It is in the IoService that the transport protocol is specified. For the TCP/IP implementation, we use a SocketAcceptor and a SocketConnector. For UDP/IP, a DatagramAcceptor and a DatagramConnector are put in place instead, even though they do not provide any functionality. In MINA, an established connection between two nodes is referred to an IoSession instance. The IoSession handles all reads and writes between to endpoints. The next component in MINA is the IoFilter. The IoFilter intercepts all events and takes the necessary actions. There can be more than one IoFilter in place. These are usually created to handle events such as event logging, authorization, thread pool, and message transformation like encryption and decryption. Finally, the last module is the IoHandler, which is where the application logic remains. All of the above mentioned components are depicted in Figure 2.7.

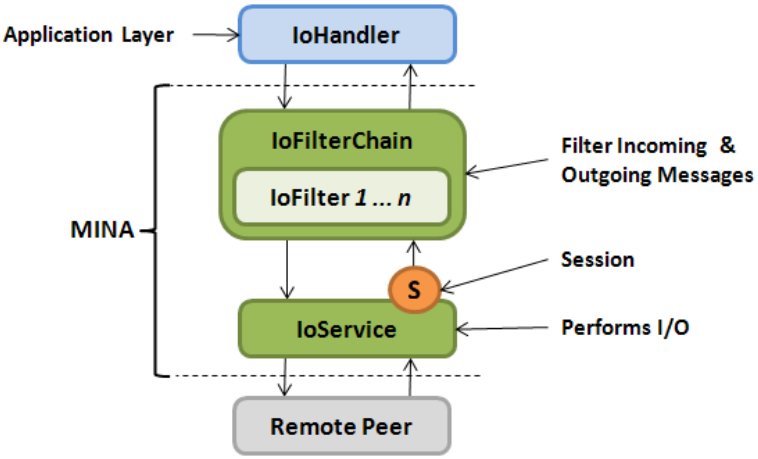


Figure 2.7: MINA Structure Overview

Chapter 3
Yobol Concepts

The aim of this chapter is to introduce Yobol. First, a general description explaining the capabilities of Yobol is given, followed by a general overview of the fundamental differences between Yobol's implementation from Mammoth and pSense as well as some challenges encountered.

3.1 Motivation

With the increasing popularity of MMOGs, using traditional client-server architecture will not suit future consumer needs as scalability is one of its biggest problem. To remedy this, peer-to-peer solutions are explored, and pSense is one of them. However, this remains a theoretical solution as no playable game has implemented it and it remains unclear what are the challenges when transferring such ideas to a real MMOG environment. Thus, it is in our interest to concretize this approach and analyze its suitability for real massively multiplayer online games. One of the goals of this thesis was the design and integration of a network engine into the Mammoth system that supports the pSense approach of position update propagation. Until now, all network engines in Mammoth were built on TCP/IP mainly for its reliability and ease of implementation properties. However, in some highly dynamic contexts, like in car racing games, where the speed of message reception is more important and messages loss can be tolerated to some extent, the use of UDP may also be

considered. In addition to the integration of this approach in Mammoth, our other goal is to evaluate its performance with different transport protocols such as TCP/IP and UDP/IP.

Essentially, Yobol is a peer-to-peer network engine implemented in the Java language. It is designed for the special requirements of network layers in massively multiplayer online games as it provides methods for direct messaging as well as functionalities for publish/-subscribe. Being a peer-to-peer solution, the idea is that in Yobol each client node holds the master object of the user's game character. The client node then becomes a server to all those who are interested in its avatar. The pSense algorithm only focuses on propagating position updates. Therefore, all other mutable objects that are not game characters will reside and be managed by the server. This makes Yobol a hybrid peer-to-peer solution where each node is equal to all others, but a server is still necessary in the system to handle all game messages not related to position updates, like player authentication and distribution, initialization of the game state and objects, and handling updates of mutable non-player objects such as a flower is picked up.

Being built on top of Mammoth, the framework underwent some considerable work remodeling and expanding the existing structure to arrive at the current implementation of Yobol. One of the fundamental changes integrating the concepts of pSense into Mammoth is to create a peer-to-peer structure. This means that some of the responsibilities previously handled by the server must be carried out in the client node instead. In addition to modifying the underlying Mammoth architecture to support pSense, some adjustments were also done on the pSense algorithm as well. In pSense all game interactions and network maintenances are merged in one system, highly coupling game semantics (position change) with message dissemination. In contrast, the design of Mammoth focuses on the separation of concerns and code modularity. We resolved this issue by creating a new component in Mammoth, called the *suggestion engine*. This engine is aware of both the game logic as well as the network engine. Moreover it is responsible for maintaining the network structure by determining which nodes one can see each other. Another issue is that the pSense algorithm only loosely describes how the connections between peers is done. To make this work in Mammoth, Yobol integrates a connection protocol into the position update mechanism proposed by pSense.

3.1.1 Naming

Yobol is a network layer API designed to handle messaging in MMOGs, whose resulting overlay structure is obtained and maintained through peer suggestions. The name chosen for Yobol reflects this main feature: *yobol* (យោបល់) is the Cambodian[1] word for "suggestion".

3.2 Challenges

In the abstract description of pSense, nodes have simple identifiers and a node can send a message to another node once it knows the identifier. Nodes get to know each other through forwarding of position update messages. In a real system, things are not as easy. First, a node needs a duplica of another player before it can do anything with it. Second, a connection between nodes must be setup prior to any communication, even in the case of UDP (a condition imposed by the Mammoth framework). All communication in Yobol is performed using the Apache MINA library. This section covers the main challenges encountered with Yobol.

3.2.1 Open Connection

Let's recall the messages in pSense. There are *position updates* from the originator to near nodes and sensor nodes and there are *forwarding messages*. In Yobol, before a node can send a message to another node, there must be a connection between the nodes. For a connection to be established, one node needs to know certain information, such as the node's IP address and port number and the unique identifier of the node, all wrapped in an instance of a *YobolNode* object. All this information allows the node to request an open connection.

[1] Cambodian or Khmer is the official language of Cambodia. It is an austrio-asiatic language influenced by Sanskrit and Pali. Please see `http://en.wikipedia.org/wiki/Khmer_language` for more information.

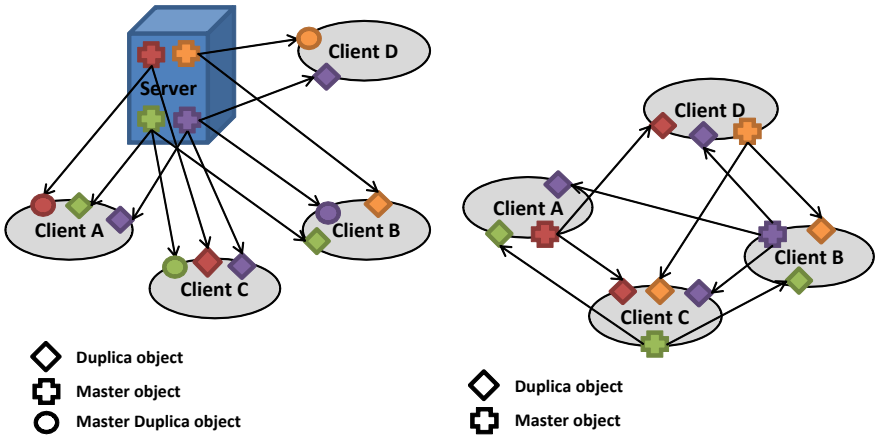
3.2.2 Suggestion Concept

pSense utilizes message forwarding to detect and connect to new near nodes, suggesting that a node A becomes aware of and connects to peer node B once A receives position update messages from B (see Section 2.5.2). In Yobol, nodes cannot communicate with each other unless they have opened a connection. If neither node A nor B has knowledge of one another, it is not possible for them to start exchanging messages with each other. Yobol tackles this by introducing the concept of making *suggestions* where connections between two nodes are made possible with the help of a third party. An arbitrary node C, which is aware of both nodes A and B, must provide the required information to either node. Node C wraps all relevant information about node A in a *suggestion message* and sends it to node B. The recipient B can then request for an open connection from A.

3.2.3 Master Object Migration

According to pSense, client nodes are to communicate to each other in a peer-to-peer manner. It does not, however, provide any further details about other functionalities found in a real game environement, like starting up the game, handling player authentication and log in, and maintaining game states. Mammoth, on the other hand, performs all these operations from its central server and its authentication service. The central server stores the master object of all mutable game objects in the game and handles their interest management among its several other responsibilities. In adapting Mammoth to support a peer-to-peer structure, some of these responsibilities are shifted from the server to the client node as seen in Yobol.

Mammoth's pub-sub mechanism dictates that to be a publisher, one must hold the master object. Initially, all master objects are kept in the central server. In Yobol, the master object of the character hosted by the client node resides in and is hosted by the node itself, thus making it a publisher. When the game's central server is started, all masters of mutable game objects are stored in it. Upon a player log in, the server sends a duplica of this client game character to the client node. The client then requests the master object of his character from the server. Upon reception of the master object, the client node has full control of the character. All other nodes wanting updates from it will need to hold a duplica object of



(a) Client-Server (b) Peer-to-Peer

Figure 3.1: Different Network Architectures Created by Migrating Master Objects

it. This makes them subscribers to this particular player character.

How master objects are distributed in the system is influenced by the type of network architecture to be used. In the case where all master objects reside on one particular node, a client-server architecture is created. On the other hand, if master objects are distributed across several machines, then a peer-to-peer structure is formed. These two cases are illustrated in Figure 3.1. In a peer-to-peer system, when a client joins, it must request the master object of the player character from the server. The server keeps a duplica of the object and migrates the master object to the client machine. From this point on, all others interested in this client's updates must send their subscription messages to the client hosting the master object. The host provides them with an object duplica, which allows subscribers to be updated of eventual changes. These steps are detailed in Figure 3.2. Client 1 hosts the master of player Bob. Client 2 wants to see Bob's movements, therefore a subscription message is sent to client 1, who in return sends a duplica back. All state changes of player Bob will be multicast to its subscribers. Keeping a duplica object in the server before the migration of the master object provides the server with global knowledge of the game world, since holding a duplica is synonymous to subscribing to the object's state updates.

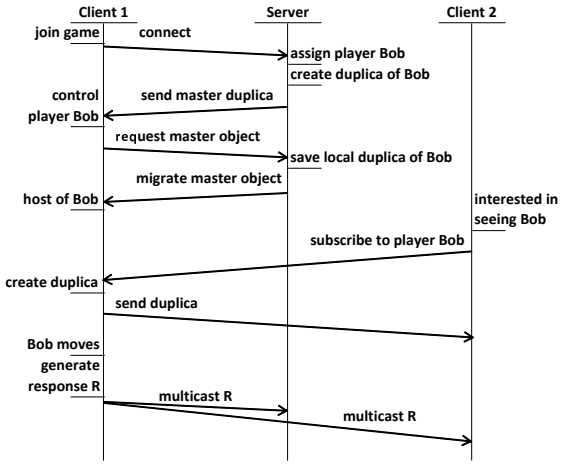


Figure 3.2: Player State Update Dissemination - Peer-to-Peer

3.2.4 Distinct Replication Spaces

Mammoth relies on its central server to perform the interest management for all mutable objects in the game. As seen in Section 2.6.4, the interest managment is handled by the replication space in the replication engine, which resides in the server. Yobol is a hybrid peer-to-peer solution where a server is still existant, but no longer performs any interest management for its player character objects in the game. Each node hosting a player character will run its own interest managmenent through a newly tailored replication space called the *Peer Replication Space*. This replication space resides in the client node itself. All other non-player mutable objects in the game are still managed by the server, therefore a distinct replication space for the server is needed, known as the *Server Replication Space*. How these replication spaces function will be explained in Section 4.4.2.

Chapter 4
Yobol Implementation

In this chapter, more detailed information about Yobol is given such as its API and the internal functioning.

4.1 Yobol Architecture Overview

The first part of this project consists of creating a network layer that supports communication in a peer-to-peer structure. This network engine must allow message multicasts, direct messages, as well as taking suggestions from neighboring nodes. The second part is to create a suggestion engine which is responsible of maintaining the overlay by emitting notifications to peers and taking appropriate actions from suggestions received from peers. The particularity of our approach is the necessity of client nodes to build their network structure themselves by receiving and sending suggestions from/to peers.

Since each component in Mammoth targets a specific functionality of the system, all procedures related to establishing and maintaining connections between nodes, as well as sending and receiving messages are implemented in the Network Engine. Moreover, all implementation regarding updating the state of an object, creating duplica objects, migrating master objects, and maintaining a list of publishers and subscribers resides in the Replication Engine. However, some operations related to the pSense algorithm such as storing and updating lists of near nodes and sensor nodes, sending and handling sensor request and sensor suggestion messages, and finding nodes for message forwarding do not fall in

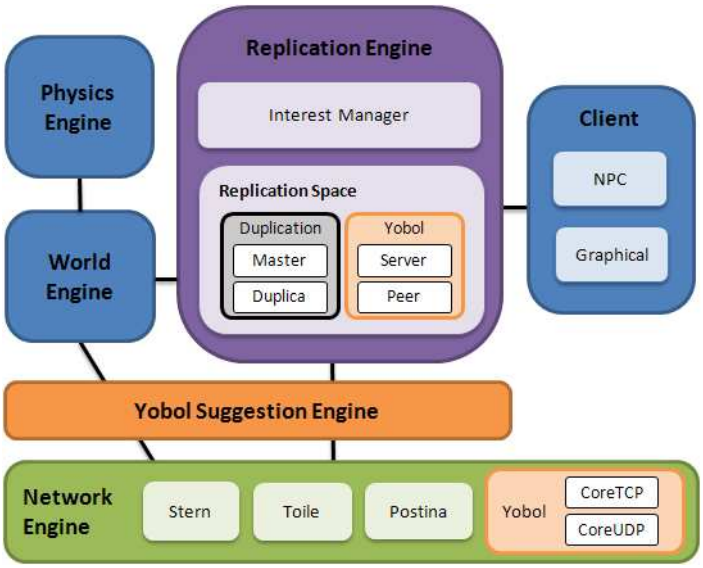


Figure 4.1: New Components in Mammoth for Yobol

any of the existing components. Moreover, in order to perform those operations, we need some knowledge about the game world. For example, to update its near nodes list, a node A retrieves the list of current connections and loops through it to determine if a peer node is to be put in the near nodes list or not. For that we need to calculate the distance between the two nodes. This distance is calculated by comparing the position of the two respective nodes, and all game objects' information, like position, are stored in the world engine. This task can only be achieved if some information from the network engine and the world engine are provided. Therefore, we have created a new component called the *Suggestion Engine*, which has both knowledge of the game world and the network engine, to handle all operations related to the peer-to-peer overlay maintenance. The advantage of using such design is the flexibility to create various suggestion engines which implement different algorithms.

In summary, Yobol adds a few new components to Mammoth: a new network engine, a suggestion engine, and a new replication space. The details about each new component follow in the later sections. Figure 4.1 illustrates their dependencies with the existing components.

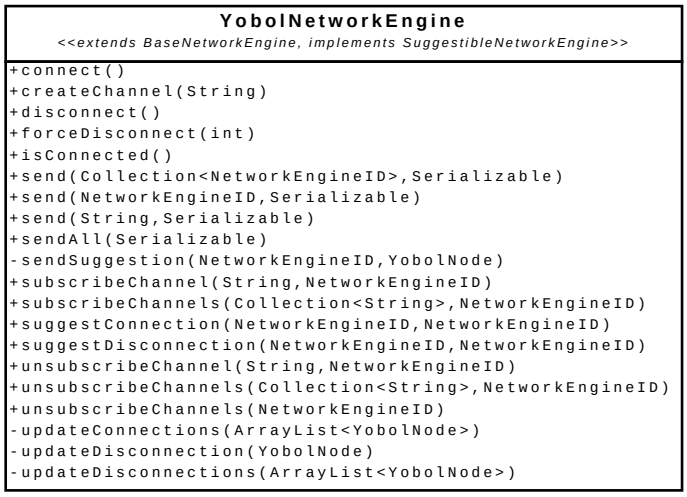


Figure 4.2: Yobol Network Engine Class Diagrams

4.2 Network Engine

Yobol's underlying network communication is built using Apache MINA (version 1.1.7) 2.8 which supports several transport protocols. This allows Yobol's network engine to be implemented in a modular way such that the underlying transport can be seamlessly switched between TCP and UPD, thus enabling us to measure and compare the system performance under each condition. Since all communication between nodes occuring in Mammoth is done using serialized objects, a custom IoFilter called the ProtocolCodecFilter was implemented in Yobol. This filter serves in translating a serialized object into a message object and vice versa.

Several other network layers already exist in Mammoth like Stern, Toile and Postina. Therefore, it is not surprising that an interface is provided by the framework to ensure modularity and facilitate future network engine development. However, the existing *Network Engine* interface did not support making connections based on peer suggestions. Therefore, a new *Suggestible Network Engine* interface is created to make this possible. The *Yobol Network Engine* implements both network interfaces. The following section provides an overview of the Application Programming Interface (API) provided by the Yobol Network Engine.

4.2.1 Yobol Network Engine API

The *YobolNetworkEngine* shown in Figure 4.2 builds the core primitive of the Yobol communication system. Since every network engine in Mammoth must implement all methods in the Network Engine interface, the Yobol network supports all methods described in section 2.6.5. Furthermore it offers a set of methods that we need specifically for the pSense algorithm. They are listed here in detail.

- **connect** has to be called first, before any other operation is possible. It instantiates an IoAcceptor object which acts as a server by waiting for incoming connections. Thus, the client can accept connections from other nodes.
- **disconnect** is used to properly disconnect the client from the network and closes all connections, freeing the ports such that they can be reused at a later time when new connections are made.
- **send** is a key method in Yobol as it is used to send a message directly to a specified client or list of clients. Other clients in the network not part of the recipients list do not receive this message.
- **sendAll** allows a message to be sent to all clients connected to the player, similar to the broadcast function.

Additionally, the Yobol network engine supports some suggestion making functionalities required by the Suggestible Network Engine interface. These are:

- **suggestConnection** informs a peer node that a connection to a specified node is desirable. This is useful to alert peers about players who just joined the game, who are currently in our vision range or who are approaching our vision range.
- **suggestDisconnection** informs a peer node to disconnect from a specific node. It is used to notify a client that a peer has left the network.

4.2.2 Message Filtering

To filter-out duplicate position updates or out-dated messages, the network engine keeps a hashtable of previously received update messages. The table stores the update message and uses the identifier of the message originator node as key. Therefore, the table contains a single update message per sender. Every position update message has a time stamp indicating when the message was created. The receiver node uses the new message time stamp and compares it with the previously received one, found in the table, from the same sender node. If the new message time stamp is smaller or equal to time stamp of the existing message, it gets discarded as the information is not *fresh* or is identical. Otherwise, if the time stamp of the new message is larger than the existing one, or if it is the first message received from that particular sender node, then it is added in the hashtable or replaces the current message from this sender in the hashtable. It is then put in the incoming message queue to be later processed.

When filtering data, only the most recent position update is kept in the incoming message queue. While old data are of some values as they provide some historical motion information which tell us about the client path from the previous known location to the current/final position, our focus was on maintaining a small incoming queue such that incoming position update messages are processed as fast as possible in order to give the client the most current game state possible.

4.2.3 Design Variation for TCP and UDP

By having a clear separation of concerns and functionalities, a more modular design takes form where each component can be replaced without causing many changes in the other dependent modules. MINA's flexible structure and Mammoth's layered architecture enabled us to implement two transport protocols to be used within the Yobol network engine without making any changes in the upper layers. Two new sub-modules were created, shown in Figure 4.1, and named *YobolCoreTCP* and *YobolCoreUDP*.

YobolCore - TCP/IP

YobolCoreTCP implements the TCP/IP protocol and uses MINA SocketAcceptor and SocketConnector as IoServices. The connection between each node is reliable and all messages sent are guaranteed to arrive at the destination node in the order they were sent out. The TCP socket receive buffer size is set to 2048 bytes. If a message is larger than this buffer size, the TCP layer automatically splits up the message into several packets and reassembles it at the destination node. Moreover, if the receive message queue of the receiver node is almost full, the sender node will modify its sending speed such that the receiver queue will not overflow and cause an out-of-memory error. Since these and several other features are provided by the TCP layer, developers do not have to take care of these things.

YobolCore - UDP/IP

YobolCoreUDP implements the UDP/IP protocol and uses MINA DatagramAcceptor and DatagramConnector as IoServices. Being a connectionless protocol, UDP provides no guarantee that a sent message will arrive at the destination. A message that is sent can be lost during transmission or arrive more than once. This means that the network layer in Yobol will have to process and filter out those duplicate messages. Fortunately, Yobol already has a message filtering mechanism in place to remove duplicate messages obtained from message forwarding. It will be more frequently used in the YobolCoreUDP implementation because it needs to remove duplicate message received due to the use of the UDP transport protocol. Another observable difference is the need to use a larger UDP socket receive buffer size. Too little UDP buffer space causes the operating system kernel to discard UDP packets when it gets full. Therefore, we set the buffer size to 16384 bytes in order to minimize the likelihood of message loss. This is performed at the cost of using more memory as well. It is a parameter that might have to be tuned dynamically depending on the game configuration.

Another challenge we encountered when using UDP is the occurrence of time-out errors. In Mammoth, when a player joins a game, it must first connect to the service server, which runs an *authentification service* to verify the player's credentials. Then it makes a

call to the *player distribution service* to retrieve the user's avatar. The client communication with those services must be done within a certain amount of time. If not, a time-out exception is thrown. Since UDP is not very reliable, the communication often takes longer than allowed. The service server then throws an exception. This prevents the user from joining the game. To remedy this, YobolCoreUDP uses a mix of TCP and UDP. When a client joins the game, it first establishes a TCP connection to the server and the service server. Afterward, all communications between client nodes are made using UDP.

4.3 Suggestion Engine

The *Suggestion Engine* is an essential component in Yobol. Its purpose is to build and maintain the peer-to-peer structure by communicating with peer nodes. It is implemented according to the pSense algorithm described in Section 2.5. The lists of near and sensor nodes are regularly updated in order to provide the client with the most accurate view of the game as possible. However, such operations can only be performed if some knowledge of the game world and the network engine are present.

4.3.1 Suggestion Making

In the sections below, three types of message are mentioned, which are *position update messages*, *request connection messages*, and *suggestion messages*. Position update messages are either sent directly from an originator node to a connected peer, or forwarded from one peer to another in order to propagate the update. Position update messages are forwarded to peer nodes that are connected to the originator node but have not received the update yet, peer nodes that have just entered the area of interest of the originator, or newly joined nodes in the game. Request connection messages are sent after a node receives a position update message from a new originator node. Request connection messages are required to gather more information about the originator node because it needs to get a *YobolNode* instance in order to establish a connection. The YobolNode object contains node specific information like the identifier, the address and port number of the node. Suggestion messages are used to transfer the required information, including the YobolNode, in order to establish a

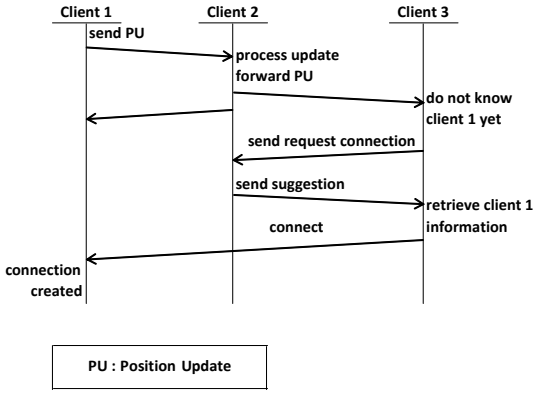


Figure 4.3: Suggesting Connection Steps

connection between peer nodes.

Position update messages contain the identifier of the message originator, the identifier of the last sender if the message was forwarded, a list of nodes who have already received the message, the number of times it has been forwarded, and a timestamp to check for freshness of information. When a peer node receives a position update message, it verifies if it recognizes the identifier of the message originator by looking in its list of connections. If there is not yet a connection, this means that it has received the message from a forward. Thus, it sends a *request connection* message to the node from which it received the message, that is the last forward. Upon receipt of a request connection message, a suggestion message containing the YobolNode object with the originator node's information is returned. This information is applied and a connection is created. In Figure 4.3, client *3* receives a position update originating from client *1* for the first time. Since the message was forwarded by client *2*, client *3* sends a request connection message to *2*, who replies with a suggestion message containing client *1*'s information. Then client *3* can initiate contact with client *1*.

These steps can be simplified by including a YobolNode object containing the information of the message originator in the update messages. However, doing so would increase the size of position update messages and this could lead to more bandwidth consumption. Moreover, using the request connection message yields a better software design because a separation of concerns is maintained. This way, position update messages are used to

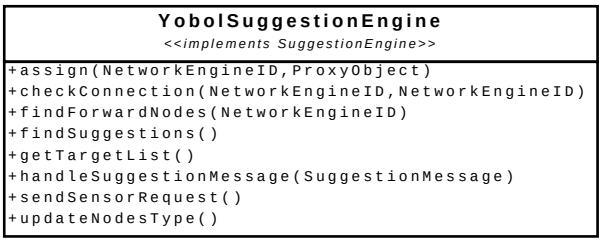


Figure 4.4: Yobol Suggestion Engine Class Diagrams

update peers about a change in players' movements and request connection messages are for building connections. Thus, better code reusability is achieved.

4.3.2 Yobol Suggestion Engine API

Figure 4.4 shows the class diagram of the suggestion engine designed for Yobol. A short description of the main methods in the suggestion engine is listed below.

- **findSuggestions** is periodically called by the replication engine to scan through the player's list of connections and see if some connection suggestions can be made to their peers.
- **updateNodeType** is periodically called by the replication engine to update the near node and sensor node lists.
- **checkConnection** is called to verify if a suggestion concerning node A and B was previously sent. This is used to avoid message duplicates.
- **findForwardNodes** is used when forwarding position update messages to peers. It finds all nodes residing in the specified player's vision range that have not received the position update message as it is the case of direct forwarding. It also finds all nodes residing close to the specified player's vision range which may know others that have not received the message yet (indirect forwarding).
- **sendSensorRequest** sends a sensor request message to its sensor nodes to request for the best sensor node candidate given a specified section. Since the sensor nodes

should be distributed as evenly as possible, each sensor node resides in a distinct area, also referred to as section.

- **handleSuggestionMessage** takes appropriate actions upon the reception of suggestion messages, which include *SensorRequestMessage* and *SensorSuggestionMessage* types.

4.3.3 Overlay Maintenance in Yobol

The peer-to-peer overlay structure in Yobol is implemented according to the pSense strategy described in Section 2.5.3. To ensure that the client's network structure is optimal according to its interests, some maintenance operations are periodically executed by the client node such as:

- **Find Suggestions:** In most cases, peer nodes discover new nodes by the mean of message forwarding as described in the pSense algorithm. Additionally to this, Yobol provides each node the capability of suggesting two distinct nodes to connect to each other. This operation is performed at every *refresh rate* determined in the replication space described in Section 4.4.

  Enabling nodes with such action could accelerate the new node discovery process. To demonstrate this, we will use the same example of Section 2.5.2, Figure 2.4(a) on page 18. A new node *P* resides in the vision range of the local node A but they do not know about each other yet. The sensor node S_1 executes its routine inspection of the connection list and detects that *P* is in A's vision range, S_1 sends a suggestion connection message to *P*. This can occur even before P receives any forwarding message from another node. Peer node *P* receives this suggestion message and opens a connection with local node A. By the time node *Q* forwards the update message to *P*, *P* already knows A and thus simply applies the object state update.

- **Update Neighbors List:** In the client-server context, a player's interest management is executed on the server. In Yobol's case, the client needs to perform its own interest management. This is done by checking who its neighbors are and saving them in the

near nodes list. Since Euclidean distance interest management is used, all players residing in the interest area are put in the near nodes list. All others are discarded.

- **Update Sensors List:** The space around a node A is split into several sections and each section should contain a sensor node. A sensor node is located a bit outside, but closest to the vision range of the node A and contained within a particular section. Each node sends out sensor request messages to each of its sensor nodes. When a sensor node receives such a message from node A, it scans through its connections list and determines if there is a peer node that is closer to A than itself, within the specified section. If such peer is found, its information is put in the sensor suggestion message and sent back to A. Otherwise, the sensor node writes its own information in the suggestion message instead and remains the sensor node for that section again.

  To find a better sensor node candidate, the current sensor node first needs to determine the section boundaries. In Figure 4.5, the sensor node is in section 1. It draws an arc using the position of node A as point of origin, and itself as the arc length. A smaller arc is drawn to represent the vision range of node A. When superimposing both arcs over each other, we are left with a hashed area, illustrated as area B in the figure. If a peer node in the sensor node's connections list is located in the area B and is the closest to node A, then it is the best sensor node candidate.

  When node A receives a sensor suggestion message, if the suggested node information is not the same as the current one, it removes the current one from the sensor list and adds the new sensor candidate to the list.

4.4 Replication Space

Yobol follows a peer-to-peer structure which requires that each client machine hosts the master object of their avatar instead of having them all on the server. Being a hybrid peer-to-peer system, Yobol still runs a server who coordinates all modifications made by mutable objects that are not of *Player* type, such as items that can be picked up or dropped down. To get the master object of their avatar, the client requests the master object from the server

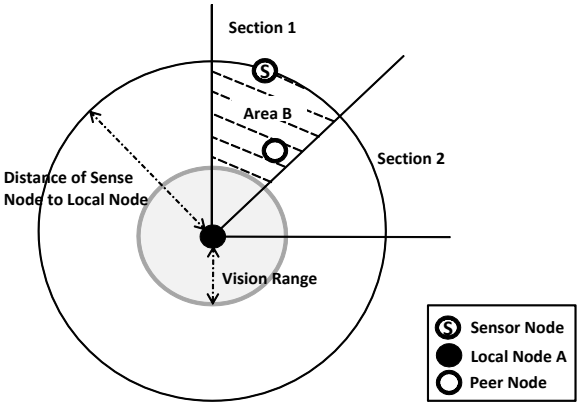


Figure 4.5: Finding a Better Sensor Node Candidate

when it is joining the game. A local duplica is stored on the server and the master object is migrated to the client. When the client receives its master object, it is declared as the master object's host. Throughout the game, new connections will be suggested from peer nodes using connection suggestion messages. When node A receives a connection suggestion message from node B, it opens a connection to B and sends a subscription message. B also sends a subscription message to A after a connection is created. After receiving a subscription message, a node adds the message sender to its subscription list and sends its own player object's duplica back. To avoid any ambiguity, please note that the subscription list contains all peer nodes currently connected to the client, while the near nodes list is a subset of the subscription list content, containing only nodes within the vision range of the client. Once they have received the respective duplica object, nodes will be informed of state updates of that object if they belong to the near node list. Figure 4.6 shows client 2 creating a connection with client 3 after receiving a suggestion connection message from client 1. Once a connection is opened, they first exchange subscription messages notifying the other party that they are interested in seeing the other entity's changes. The operation is completed when both clients have received the other client's object duplica.

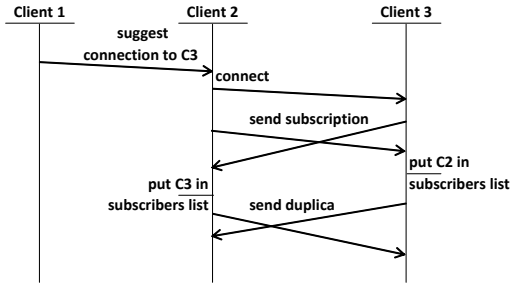


Figure 4.6: Steps Taken to Establish Connection Between Peers

4.4.1 Refresh Interval

Most operations taking place in the replication space are performed periodically. Each period is determined by the refresh interval value, which is the amount of time elapsed between each execution of the overlay maintenance on the client node. It is a parameter that might have to be adjusted according to the game configuration. In the current setting, players make a move every 500 milliseconds. The refresh interval value is set to 15 milliseconds. It is more frequent than the movements of a player because the replication space needs more information in order to set up and maintain the peer-to-peer overlay structure. The replication space performs the following actions for each refresh interval:

- verifies if the client master object is migrated or not (only done at initialization time)
- scans through the client's list near and sensor nodes and checks if it can suggest a connection should be made between two nodes
- updates the client's near and sensor node lists according to other clients new position updates and sensor suggestion messages received
- sends sensor request message to find better sensor node candidates (performed every 5th refresh interval)

The chosen refresh interval value plays a major role in determining how fast a player can discover other peer nodes in their vision range and finding better sensor node candidates. The refresh interval value must be chosen with care because if it is too large, the client can send its update messages to a stale list of client nodes where some of them may have already

left the client's vision range. If the value is too small, the client will perform too frequent updates on its lists of near and sensor nodes. This leads to higher resource consumption on the client node and an increase of traffic on the network due to the messages exchanged to find better sensor node candidates.

4.4.2 Replication Space: Server and Peer

Since clients can only see other players if they have their object duplica and duplicating every game object is usually inefficient, a *replication space* is implemented to manage object replication in the system (i.e. which client machine should get a duplica of which object). Section 2.6.2 describes a replication space that implements the distributed object model. This replication space is located on the server where a matching function is periodically executed on pairs of publishers and subscribers to let subscribers discover new publishers. The server performs these checks because it has all master objects and has a global knowledge of the game world, which makes it the best candidate to make such decisions. In Yobol, the master objects are hosted by their respective clients, such that each client maintains the list of subscribers interested in the movements of the client's game character. A new replication space called *Replicaton Space Peer* is created to be run on every client node. It is a space containing all duplicated objects that reside in the vision range of the client's avatar. The Yobol server also has its own replication space known as *Replication Space Server*. Its usage is limited to assigning a duplica to a newly connected client, migrating master object to a client, keeping the state of local duplicated object up-to-date, and maintaining master objects of items.

As the name suggests, the Replication Space Server resides on the server node and is used to provide a duplicate of the avatar of a newly joined player to its client node. After receiving its character's duplica, the client requests the master object from the server, in order to enable the client to host its own game character. This *Request Master* message is of *Replication Space Message* type such that it will be handled by the replication space at the destination, which is the server in this case. The request master message contains information about the Player object whose master object needs to be retrieved. The server replication space receives the request master message, keeps a duplica of the object in order

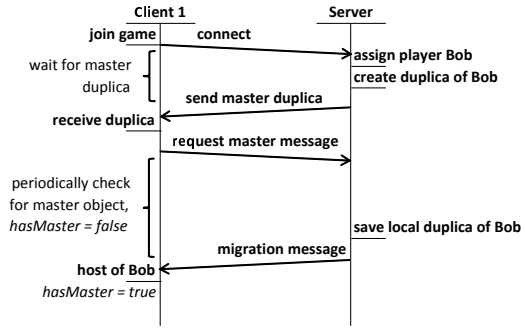
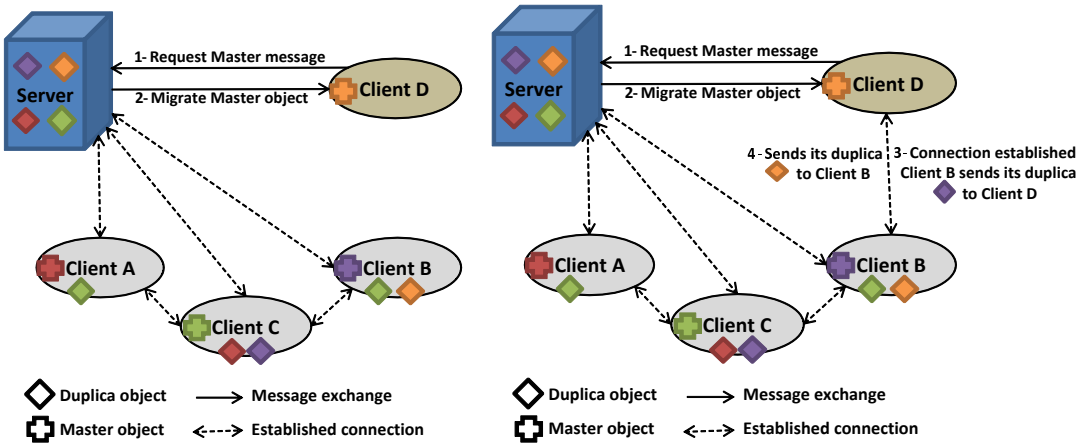


Figure 4.7: Steps Involved in Replication Space Server and Peer

to remain updated about its later changes, and proceeds with the generation of a *Migration Message* which is sent back to the client.

On the client side, a periodic check is performed to verify whether or not it has received the master object of its avatar. During the initialization time of the replication space peer, a boolean flag is created to indicate if the master object migration process has already taken place. It is initially set to false, and will be changed to true only when the client receives the Migration Message from the server. The client becomes the host of the master object, and thus is the publisher of that object. The interaction between these two replication spaces is depicted in Figure 4.7 where the Replication Space Peer is run on the client node and the Replication Space Server on the server. No resent of the request master message is done if the client fails to receive its master object. In the case of a failure, a time-out occurs on the master request forcing the client to exit.

From this point on, players who are interested in the client's movements will subscribe to it in order to receive its publications. Clients can only see other players if they have their object duplica. Therefore, when new connections are created between two nodes, they notify their mutual interest by sending a subscription message to each other. When a subscription message is received, the client creates a duplica for each newly connected client node. This gives control over which client machines will get which object duplica. In Replication Space Peer, a player object is both a publisher and a subscriber. A publisher object disseminates its own state updates, whereas a subscriber object finds new publishers and subscribes to them in order receive their updates. Players in MMOGs generally



(a) client D joins the game (b) exchange of duplicas

Figure 4.8: Master Object Migration Steps

control only one player at a time during a game session. Therefore, it is reasonable to limit each client node to have only one master object of a player at any instance in the game. Subscribers discover new publishers only when new connections are created. There is no matching policy used in this replication space. Therefore, in order to detect new connections, the client replication space must periodically invoke some operations in the suggestion engine which will trigger the execution of some maintenance of the peer-to-peer structure, like finding new connection suggestions for the client's peers, updating its near nodes list, and updating its sensor nodes list by sending out sensor request messages and handling their response (see previous Section 4.3). Figure 4.8(a) illustrates an example where a newly joined node D requests the master object of its avatar from the server, since all masters reside on the server initially. Once D has received the master object, it can start processing client B's connection request and exchange their duplicas, in order to get updates about each other. B must send its duplica to D, and vice versa, based from the assumption that the interest relationship between players is symmetrical: *"if B is interested in D, then D is also interested in B"*. This is illustrated in Figure 4.8(b).

4.4.3 Object Migration in Yobol

Burst Migration is the current migration support implemented in Mammoth, where the sender of the master object can initiate a migration at any given time without previously notifying the receiver end that it wants to migrate some objects. However, in our approach, object migration is not so sudden anymore because the client initiates the contact with the server first in order to start the object migration procedure. Therefore, clients do not have to worry about suddenly receiving a master object. The client notifies the server that it wants to host the master of a given game object using a request master message. The server receives this request message, proceeds with setting the local copy to a duplica, and marks the client id as the master object's host. This information is wrapped in a Migration message and sent back to the client. This allows better control over the object migration procedure.

4.5 Peer Communication Strategy

The creation of object duplicas is managed by the replication space contained on the server or the client node. To maintain the list of subscribers and publishers and abstract the object replication logic from the network layer, another sub-module is added between the replication engine, the suggestion engine, and the network engine. It is called the *Peer Replication Strategy*. Its purpose is the same as for the communication strategy mentioned in Section 2.6.5, but in this peer replication strategy some of the existing methods were overridden to adapt to the needs of the peer-to-peer structure and logic of Yobol. By adding another layer to the structure simplifies the amount of work should we decide to change the communication logic. A new implementation can be dropped in and replace the current one, without requiring much or any modifications to be done in the other modules/engines. The modified methods are:

- **publish:** is used to send a state update to all clients subscribed to a specific topic. The pSense algorithm introduced the idea that client machines may not be able to send position updates to all nodes located in their vision range due to a limited amount of upload bandwidth capacity. Messages are then sent to a subset of players in the

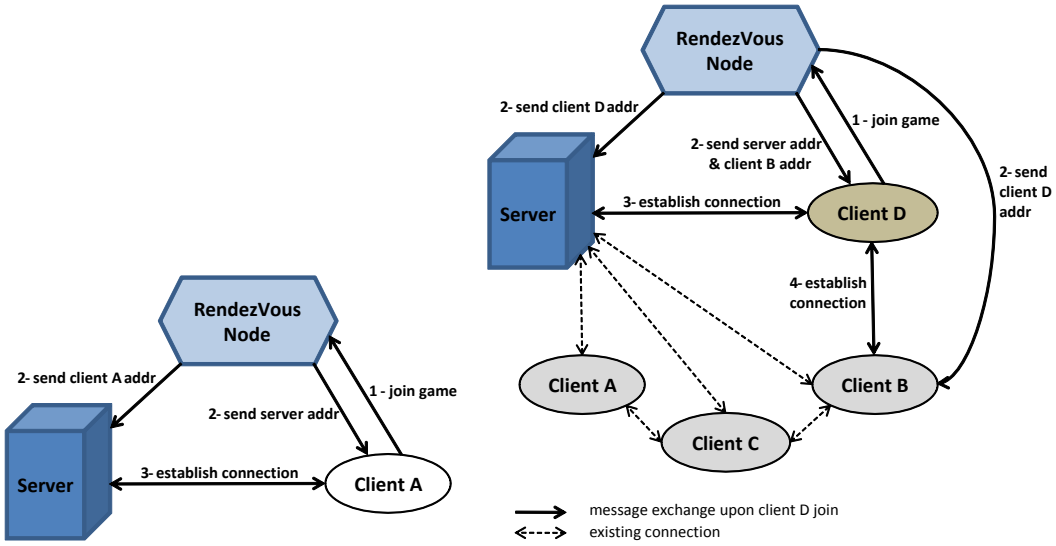
vision range. The others will receive the position update through message forwarding (see Section 2.5.3). Yobol follows this approach by creating a *MAX_OUT_CAPACITY* parameter which limits the number of messages a client sends per position update. By default, the position update is to be sent to all clients whose identifier is in the near nodes list. However, if this list is larger than the MAX_OUT_CAPACITY value, a copy C of the near nodes list is made. The contents in C are shuffled and the list is shrunk to reach the size defined by the MAX_OUT_CAPACITY parameter. The recipient list defined in the position update message is filled with the identifiers contained in C.

- **sendToTarget:** is used to send a message to a target client. With the possibility of sending position update messages to only a subset of peers due to the limit imposed by the MAX_OUT_CAPACITY value, the existing *sendToTarget* method (see Section 2.6.5) needs to be altered in order to enable such operation.

4.6 Boostrapping

In a common client-server setting, clients must first connect to the game server in order to be authenticated, to retrieve the latest state of the game world and of their avatar, such that they can continue playing where they last left off. This task has proven to be more challenging in the context of a peer-to-peer structure, especially when no superpeer node is used. With no superpeer nodes, no nodes have a global knowledge of the game. Therefore, when a client joins the game, no one can efficiently determine to whom it should connect to. Note that boostrapping occurs before any game interaction takes place.

In Yobol, a *RendezVous* node was created to facilitate initial connection set-up. The RendezVous is itself a server node and contains information which helps a client A locate the game server and an active client B currently connected in the game. Client A will then connect to the server, request its master object and connect to client B. Once A is connected to B, A collects information about the current game state from B and builds its near and sensor list. The RendezVous node is kept separated from the game server node because each entity focuses on a different concern, and separating these concerns is favorable for



(a) first client joining (b) fourth client joining

Figure 4.9: RendezVous Node Bootstrapping

code reuse and maintenance. Moreover, this allows server access to authenticated clients only, which is desirable for security reason.

In more detail, to join a game the client must first connect to the RendezVous node. Then it waits for a reply message from the RendezVous, which contains the game server's and the service server's addresses. Once the client is deemed valid, it receives an avatar from the service server and the game states (duplicas) from the server. It then loads the game map. At this point, the client only holds a duplica of his avatar. The next required step is to request the server to migrate the master object to the client. A peer-to-peer structure is then formed.

When the RendezVous node finds a node that has not been connected to the game server yet a new connection is detected. Therefore every 5 seconds, the RendezVous node scans its connection list looking for any node whose flag is set to false. This means that a new client node has connected. Once the node is connected to the game server, its flag is updated to true, thus indicating that it is an existing node. When a new connection is found, the RendezVous node forwards the client's address to the server and the authentication service. If this new node is the first client to connect, then its address is forwarded to the server

and services only. Otherwise, it also forwards the new client's address to a random client who is already connected to the game. These steps are illustrated in Figure 4.9. In the case (a), there are no previous connected nodes in the network, therefore client A's address is only forwarded to the server and vice versa. In (b), client D wants to join the game and the RendezVous node chooses client B as the node which client D will first connect to. It is through client B's connections that client D will build the near nodes list and sensor nodes list. When a client leaves the game, the RendezVous node receives a *disconnect* event and proceeds to remove that node from its connection list.

Chapter 5
Experiments

We have run a suite of experiments using different plausible game scenarios. Below we describe the context in place, and present an analysis based on the various data gathered during the experiments. These discussions demonstrate both the suitability of Yobol as a network engine for MMOGs, and also how the system is affected by using TCP/IP or UDP/IP as transport protocol. The experimental environment used is detailed in Section 5.1. The simulation setup evaluated is presented in Section 5.2 and the results are summarized in Section 5.3.

5.1 Experimental Environment

The implementation of Yobol has been extensively tested inside Mammoth using clients with Non-Player Characters (NPC). The behavior of the NPCs used is rather primitive as the NPCs are simply moving around in a random pattern in the game map. However, this behavior is perfectly appropriate for our interest as it causes many messages to be sent over the network. This uses both direct messages (to tell the master that the client has moved) and published messages (to inform other players about the change). Also, as the NPC moves around the map, other node discoveries and disconnections can be observed.

The NPC-clients were executed on 30-50 computers from the McGill School of Computer Science computer labs. Each of these machines is equipped with a processor that runs at a minimum speed of 2GHz and with at least 2GB of memory. A small script is

used to remotely log in to the machines and then start the NPC clients. Several NPC clients can be run on one machine to simulate higher load settings. A 5 seconds delay is added between the startup of each client to avoid a bottleneck on the RendezVous node. Since in real MMOGs, players rarely connect all at the same time, this is reasonable. Being a hybrid peer-to-peer system, Yobol's server is run on a machine with 2.GHz processor and 8 gigabytes of memory. We use the same virtual game map for all our experiments. This 30 x 30 size map contains about 500 player objects. This is the maximum number of clients that can connect to it.

5.2 Simulation Setup

In the simulation setup, each NPC has its own environment and runs in its separate thread. This means, when a new NPC connects to the server component, it receives a copy of every object it is interested in. It can then interact with the world by executing actions on those objects.

Communication between NPCs and the server component is achieved using the *Yobol-NetworkEngine*, which is a real socket-based communication engine. The central server and the NPCs run on different machines. Each NPC has to spawn at least 2 threads, one to communicate with the centralized server, another to monitor the sockets for incoming traffic, and additional ones for each node connected to it. In addition, all communication is routed through the localhost network interface. Each machine may be running one or many NPCs depending on the experiment at hand.

The interest area of the NPC is set to 7 such that each player character sees about 17% of the game map. With a smaller vision range, the number of connections between nodes decreases. However, the usage of sensor nodes is essential to support a peer-to-peer overlay without causing any network partitions. No limit in the number of outgoing messages is defined in our experiment such that any player can send out as many position update messages as needed after a movement. However, some message forwarding is still expected to be observed as it is used to detect new near nodes in the network. The same setup is used for all experiments with the exception of the underlying tranport protocol

(TCP/IP or UDP/IP) used in the network engine.

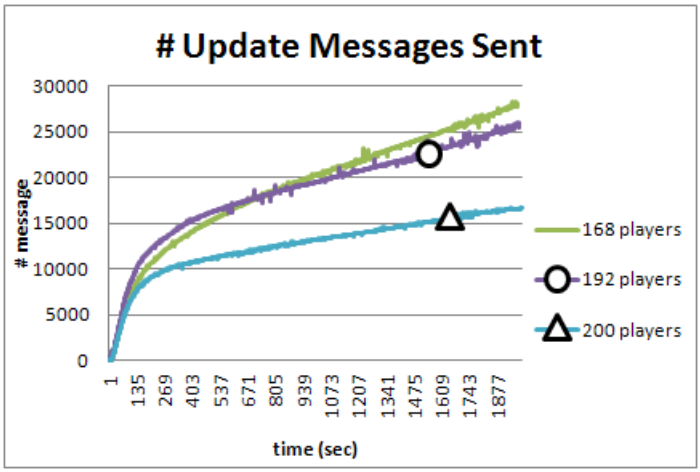
5.3 Results

This section provides some data collected from our experiments and their analysis. To determine the feasability of using Yobol in a real game environment, some measurements such as the connection latency of player nodes and determining the maximum capacity of each node are performed.

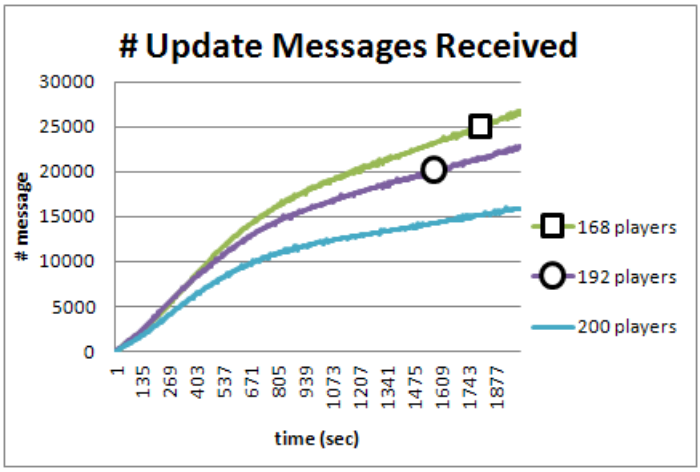
5.3.1 Capacity

Our initial intention was to compare the scalability of the peer-to-peer system to the existing client-server structure. However, largely due to a lack of computing resources, we were not able to generate enough players to make this analysis interesting. From the available computers in the laboratories, only a subset of 30 machines was suitable for our experimentations. As many of them are older machines and contain less memory, they quickly become overloaded especially with the TCP/IP implementation of Yobol.

Each socket connection uses one file handle. As the number of connections per client increases, the file handle size also increases, up to a point of running out of file handles. This means that no more files can be opened, no more socket connections can be accepted, and no more shared libraries can be loaded. This would result in a denial of service scenario where no more connections are allowed, and under some circumstances, the game client node can hang or crash. A simple work around is to increase the file handle size. However this comes with a cost, as overallocating carries a penalty of cost (memory and/or cpu). The default value of file handle in Linux is 1024. In order to allow clients to keep a larger number of connections, without letting it grow out of hand, we have limited the size of this handler file to 51200. Despite our efforts, only 3 NPC clients can be simultaneously run in a machine without causing it to overload. Therefore, the TCP/IP implementation is only able to scale up to 90 clients randomly distributed over the game map. This does not signify that the system is not scalable with TCP, but this limit was reached because many clients are run on each machine. Generally each machine hosts only one client.



(a) Total Number of Position Update Messages Sent



(b) Total Number of Position Update Messages Received

Figure 5.1: UDP Maximum Load

The UDP/IP implementation of Yobol requires less computing resources because it is a connectionless protocol. Therefore, more clients can run at the same time on a machine. For the TCP experiment, some machines that were not suitable to be used due to their lack of memories, can now be used for the UDP experiments. This increases the number of machines available to 50. Our experiment demonstrates that the server becomes overloaded when more than 300 clients connect to it and a client machine has the capacity of running 7 NPC clients using UDP. However, our system was only able to scale up to 168 clients randomly distributed in the game before service deterioration is observed mainly because several clients are running on the same machine, forcing it to run out of computing resources. Figure 5.1 (a) and (b) respectively illustrates the total number of messages sent during the game by all players, and the total number of messages received. The x-axis is the time in seconds, and the y-axis is the number of messages. During initialization time, there are more position update sent out than received. Once the system is stable, the number of messages sent and received increases linearly because players publish their new coordinate regularly to their peers. These graphs show that the largest amount of message sent and received in the game is achieved with 168 players. As the number of clients increases, the number of messages gradually decreases. This clearly indicates a degradation of service due to high network traffic. In the peer-to-peer architecture, each client sends more messages. Therefore our client machines saturated faster. Note that in a true peer-to-peer system, this problem does not arise because each machine only hosts one client.

From this experiment, we can conclude that TCP uses more resources than UDP as in our setting, TCP only scaled up to a total of 90 players (3 active clients per machine), whereas UDP supported up to a total of 168 players (6 active clients per machine). In both cases, the system was overloaded because too many clients were running on the same machine, and more time was spent on transmitting and receiving data packets instead of processing them.

Let's recall that TCP performs some end-to-end flow control such that data are not sent faster than the receiver is able to receive and process it. TCP breaks large messages into smaller packets and guarantees that every packet sent arrives at its destination in the same order as it originated. Doing this generates a considerable amount of overhead. This explains why only 3 active clients can be hosted by one machine with TCP. Being a lighter

protocol, UDP uses less resources as no hand-shaking is required prior to sending a message. This can be observed by the fact that each machine can hold up to 6 active clients. By putting more than 6 clients on one machine, a lot of stress is put on the machine, which made it unable to process and perform its clients tasks. Overall, we cannot judge that UDP is more scalable than TCP as both have hit a limit which occurred because each machine hosted too many clients at a time.

5.3.2 Performance Comparison: TCP VS UDP

In this section we describes the experiment performed to analyse the performance of TCP and UDP with Yobol. The experiment is run 3 times for each set of 30, 60, 90, and 120 randomly distributed NPC clients. The measurements are taken over a time span of 20 minutes of game play. Our analysis focuses on four criteria: the amount of time requires for a client node to connect to a peer node, the amount of CPU used by the client machine and by each client process, the memory usage of a client process, and the amount of occured page faults.

Connection Latency

In Yobol, a local node discovers a new peer node by the use of connection suggestion messages. Our intention in this first experiment is to measure the latency for creating a connection after the reception of such message. When the local node receives a suggestion connection message, the current time stamp and the new node identifier is stored in a table. The local node establishes the proper connection with the peer node and waits for the latter's replica. When the replica is received, another time stamp is generated. The difference between both times determines the connection latency. Figure 5.2 illustrates the latency observed when TCP or UDP is used as transport protocol. We can observe that it takes more than 1 second to create a connection between client nodes. This is because the connection latency measured here includes the amount of time taken for a client to request and receive a duplica object.

Figure 5.2 TCP has a small latency which is comparable or sometimes better than UDP when the system is not overloaded like with 30 and 60 clients. At 90 clients, the lack of

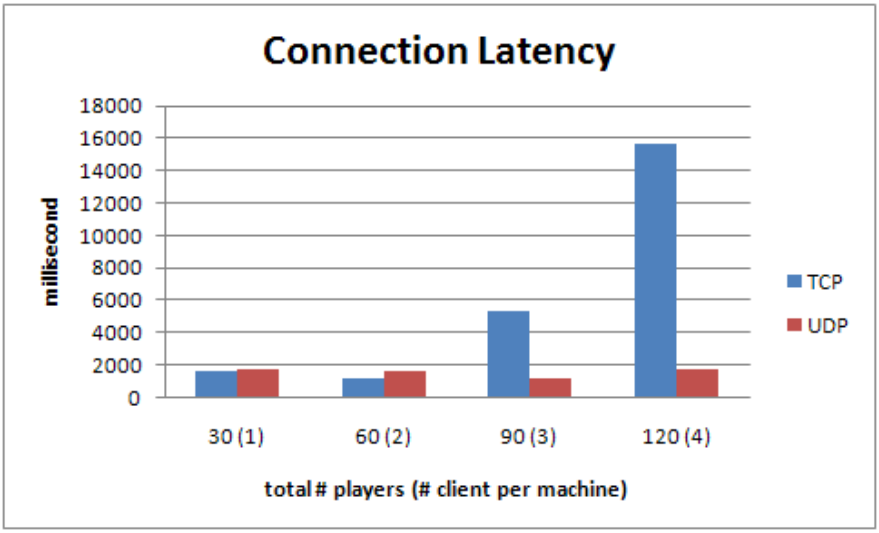


Figure 5.2: Connection Latency Comparison

machine resources is starting to put a strain on the clients' execution. Each client needs to hold more connections and receives much more messages. Being run over TCP, if a client becomes overburdened by the incoming messages and starts taking more time to read, the sender will adapt its sending speed to the reading speed of the receiver. If the message being slowly sent is a connection request, then a large latency will be measured. In the case of UDP, no connection needs to be established before a message can be sent. Messages are sent out without any guarantee of being successfully received. UDP does not adjust the sender's write speed to the receiver's read capability as done by TCP. Since no handshake needs to be done between two parties before they can start exchanging messages, and even with high traffic on the network, nodes can perform a write as fast as they can support, UDP performs much better than TCP under high loads. While TCP is struggling with the lack of computing resources, UDP shows a very small variation of its latency, which is considerably less than observed with TCP.

Although TCP performs well in under-loaded situations, the latency quickly increases when client nodes become overloaded. Moreover, the system becomes overloaded with only 90 connected players. While this can confirm that TCP has reached its maximum capacity with 90 active players, UDP's scalability does not seem to be affected as its latency

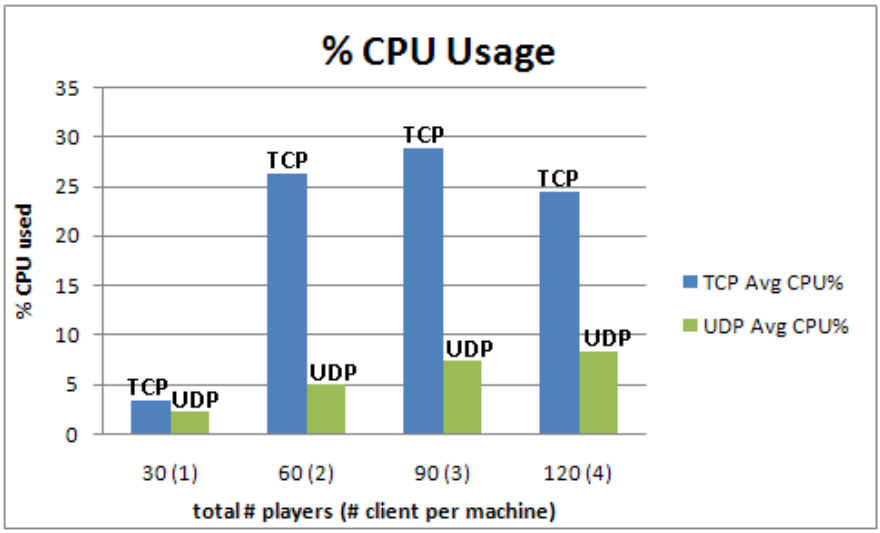


Figure 5.3: Performance Comparison - CPU Usage

remains under 2 seconds throughout the experiements.

CPU Usage

Figure 5.3 displays the experimental results for the average amount of CPU used to execute the application code in the user space (for example programs and libraries). When comparing the system performance running over TCP and UDP, in every case, TCP needs more CPU than UDP. As the number of clients in the game increases, a local node A needs to maintain more connections with near nodes. Therefore, updating its near node list and sensor node list will take more time because a node A will have to go through a longer list to determine whether a node B should be in its near list or not. The CPU usage for TCP peaks up at 90 connected clients and suffers a significant drop afterward. This confirms that the maximum capacity of the TCP implementation is indeed 90 players, because when more players try to connect to the game, the system becomes overloaded and some of these connections are dropped, causing a decrease of CPU usage for all remaining clients.

Although TCP uses more resources for keeping connection sockets than UDP, the latter needs to perform more work in the application layer to compensate for some of the lacking features of TCP, like message duplication. As mentioned in Section 2.7, in TCP a message

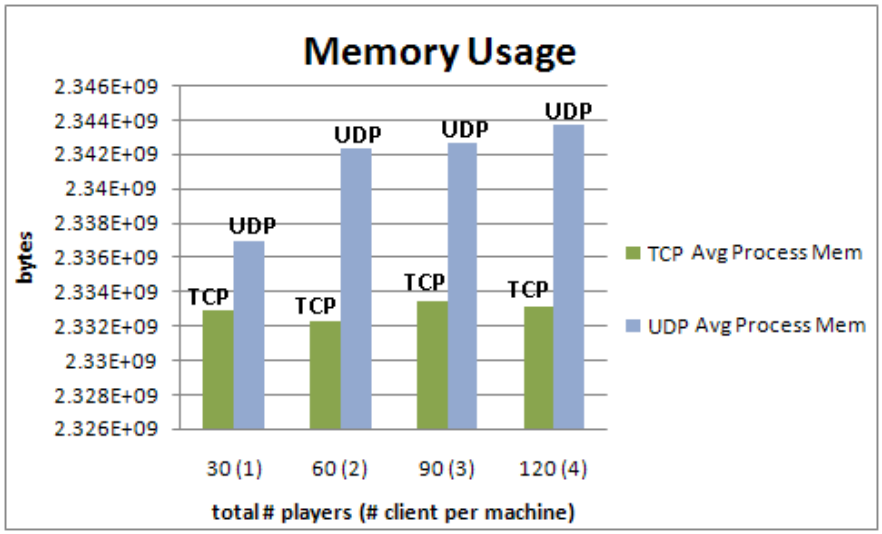


Figure 5.4: Performance Comparison - Memory Usage

is either sent once or not at all. UDP does not offer this guarantee such that a message may be sent zero or many times. In order to provide only relevant data to the game layer, our UDP network layer must filter out more duplicated messages. This is reflected by an increased usage of CPU as the number of clients grows larger for the UDP transport protocol. Nevertheless from the above figure, we can judge that UDP's scalability limit of 168 players is not caused by a lack of CPU power as its CPU usage remains well under 10% throughout the experiments.

Process Memory Usage

The average process memory usage of Yobol running on TCP is significantly lower than on UDP. This difference is due to the use of a larger socket receive buffer size for the UDP implementation. Too little UDP buffer space causes the operating system kernel to discard UDP packets. To minimize the likelihood of message loss, the buffer size is increased to 16384 bytes (2^{14}) for the UDP implementation. The TCP socket receive buffer size is only 2048 bytes (2^{11}). For each datagram socket created a greater amount of memory is allocated for the reception of messages. The average use of virtual memory per process for the TCP and the UDP implementation of Yobol is depicted in Figure 5.4.

For TCP, the amount of memory used fluctuates between 2.332GB and 2.334GB as the number of player increases. This is mainly because TCP uses a smaller receiver buffer size than UDP. In the case of UDP, as the number of clients increases, the amount of memory used increases as well. A significant growth is observed when going from 30 to 60 players (i.e. from one player to two player per node). An increase of connections between client nodes can account for this. However, a smaller augmentation of memory usage is shown for 60 to 120 players. This can be explained by the fact that the vision range of client nodes covers about 17% of the map and many groups of clients, located in different areas in the map, can be formed. If a client belongs to one of those groups, the client's connections are limited to the peer nodes in the group along with a few sensor nodes. This is an effect of an under-populated map. In this case, the number of connection held by a client at a given time would not vary much.

From the above figure, we can determine that with a total of 120 players in the game, there are 4 clients running on each machine. Each client process consumes on average about 2.343GB of virtual memory. Since each machine contains 8GB of memory, it is not surprising that UDP hits its scalability limit of 168 players (6 clients per machine). Once more, this limit is reached because more than one client is executed on one machine, which is not common in a true peer-to-peer game environment.

Page Fault Count

From Figure 5.5, the occurrence of page faults is very similar for both TCP and UDP under normal load. As TCP reaches its maximum capacity of 90 players, a greater amount of page faults ensues. Combining this finding to the result obtained in the connection latency section, we can safely conclude that the system is overloaded from this point on. However, Figure 5.4 shows that the memory usage for TCP is low but Figure 5.5 shows that there is a large number of page faults. An explanation for this is that the system may be thrashing. In our TCP implementation, we have used the default flow-control provided by MINA, providing the basic functionality. This flow-control may not be adequate for Mammoth's game environment as illustrated in the obtained results. With 90 players in the system, each client has reached a point where it is unable to process all incoming

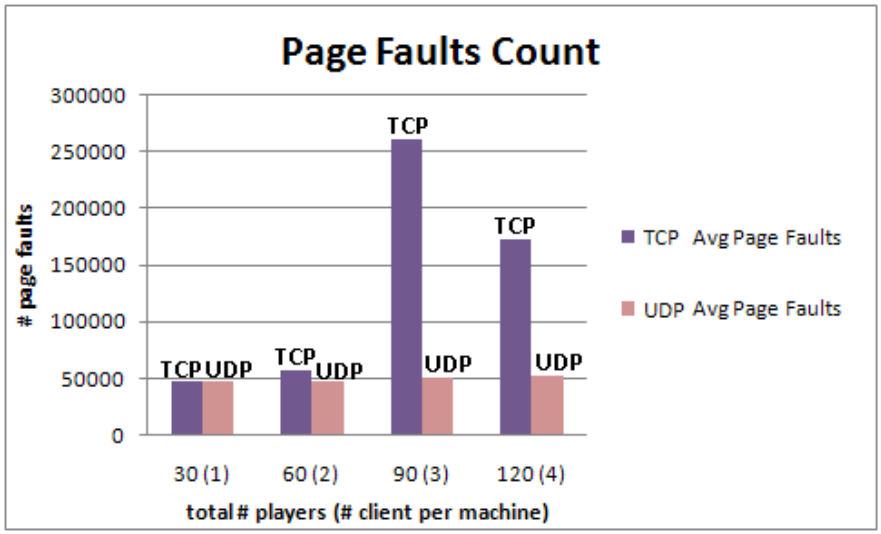


Figure 5.5: Performance Comparison - Page Fault Count

data. Therefore, it requests that its peers reduce the amount of data they send at a time by lowering the window setting value on a TCP packet. If the client is still unable to process all incoming data, this window setting value becomes smaller and smaller, to the point where the data transmitted is smaller than the packet header itself. This makes the data transmission extremely inefficient. Since there is a certain amount of overhead associated with processing each packet, the increasing number of packets means an increase overhead to process a decreasing amount of data. This results in a thrashing system, which results in a large amount of page fault occurrences.

Since the value shown in Figure 5.5 is an average value of all experiment runs, TCP could be running fine for 95% of the time, then at some point something causes all the memory to be used up for a few minutes and causing thousands of page faults. This will have very little affect on the memory usage analysis, but a large one on the page fault statistic.

Message Overheads

Aside from sending and receiving position update messages, each client node is required to communicate a set of messages that are essential to the functioning of the game. These fall

into the message overhead category. Some of the observed overhead messages are listed below.

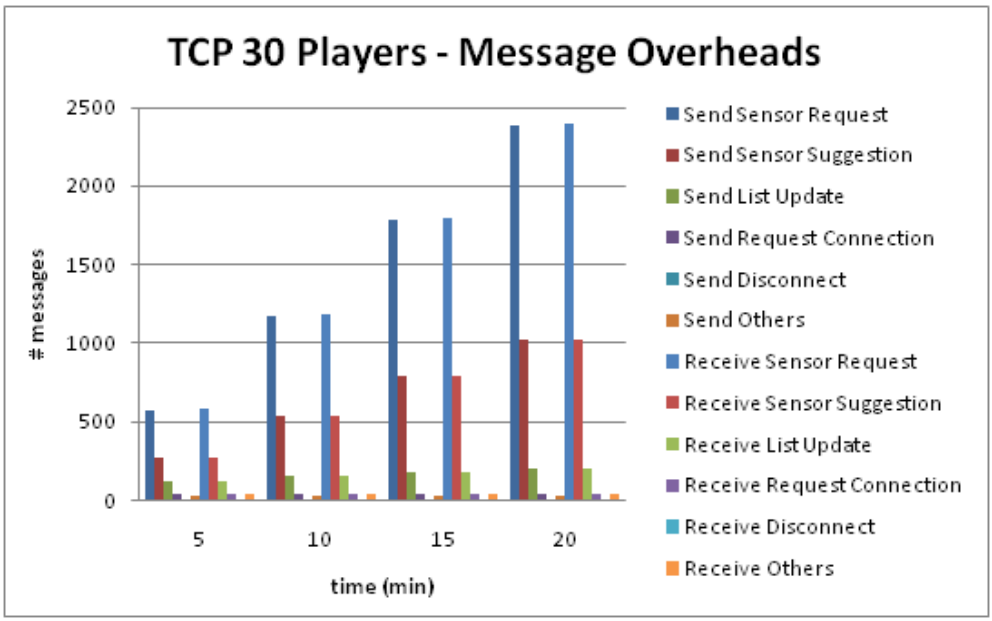
- **Sensor request messsage:** sent to sensor nodes asking for the best sensor candidate in a given area/slice
- **Sensor suggestion message:** sent by sensor nodes indicating which node is the best sensor candidate for a given area/slice
- **Request connection message:** to ask a near node B for more information about another node C such that it can initiate a connection to C
- **List update message:** to create a connection with new node C (once this message is received, a concrete socket will be opened on both nodes to allow data transmission)
- **Disconnect message:** to notify that a near node B left the game

Figures 5.6, 5.7, 5.8, and 5.9 illustrate these overhead messages in four different setting: with 30, 60, 90, and 120 players in the game, using TCP/IP as transport protocol or UDP/IP. The y-axis represents the total number of overhead messages occuring after a given time has elapsed. The x-axis is the amount of elapsed time.

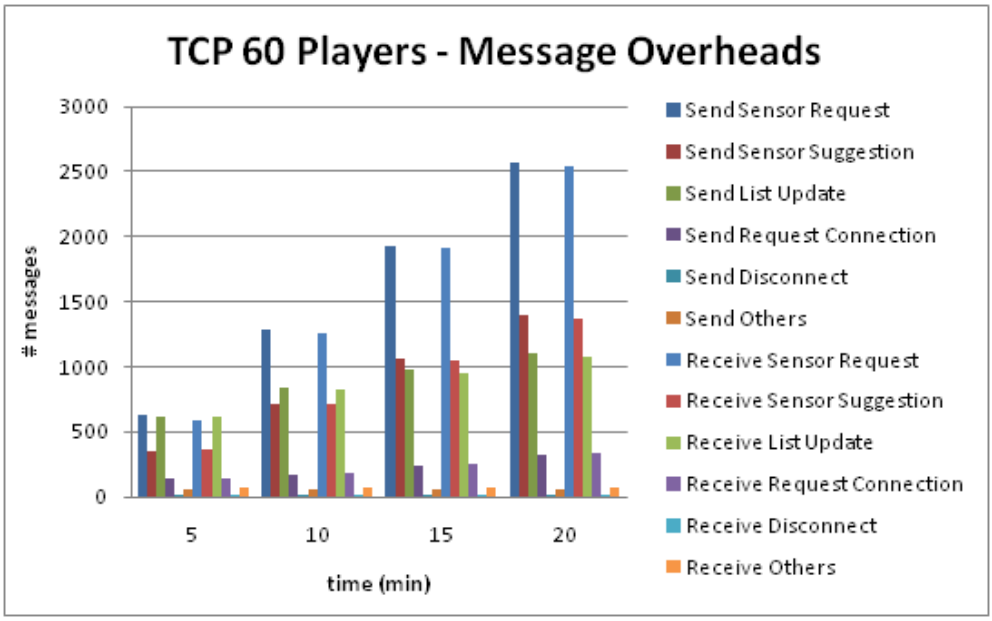
When comparing Figure 5.6(b) with Figure 5.7(c), we can observe that the number of list update messages increases significantly with 90 active players in the system, while the number of sensor request messages decreases a little. This conforms to the fact that more connections are needed as the game map becomes more populated, but it also suggests that more computing resources are used to create and handle connections with peers rather than performing overlay maintenance. At 120 players (see Figure 5.7(d)), TCP is already overloaded, therefore the number of overhead messages is lower than at 90 players.

When comparing Figures5.8(e-f) and 5.9(g-h), we notice that as the number of players increases, the amount of sensor request messages decreases a little, while the number of request connection messages increases greatly. This indicates that more effort is put on trying to create connections with new peers than maintaining the network overlay. Another point worth mentioning is that the number of sent messages is expected to be greater or equal

to the number of message received for each type of overhead message as some messages can be lost. As anticipated, the results collected for TCP abides by this. This is because TCP is a reliable transport protocol such that every data packet sent out is guaranteed to arrive at its destination exactly once. In contrast, a new trend is observed for UDP where the number of message sent is less than the number of message received, as the game map becomes more populated. This can be the result of message duplication in UDP. Let's recall that UDP does not provide any guarantee that a message sent will be successfully received at the destination. Also, a message can be sent out more than once, such that the same message is received multiple times, but was counted as a single sent.

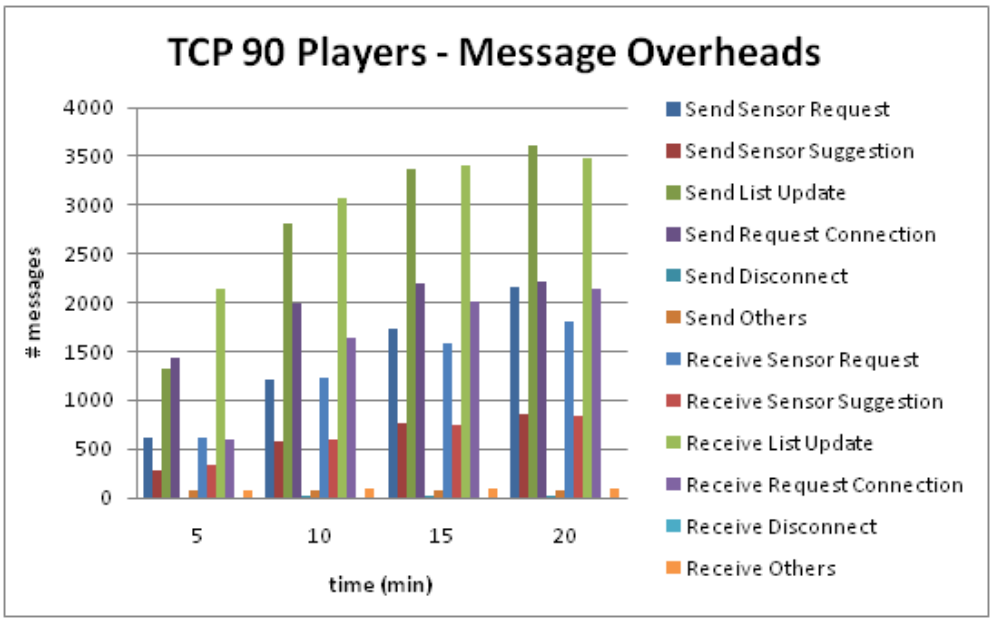


(a)

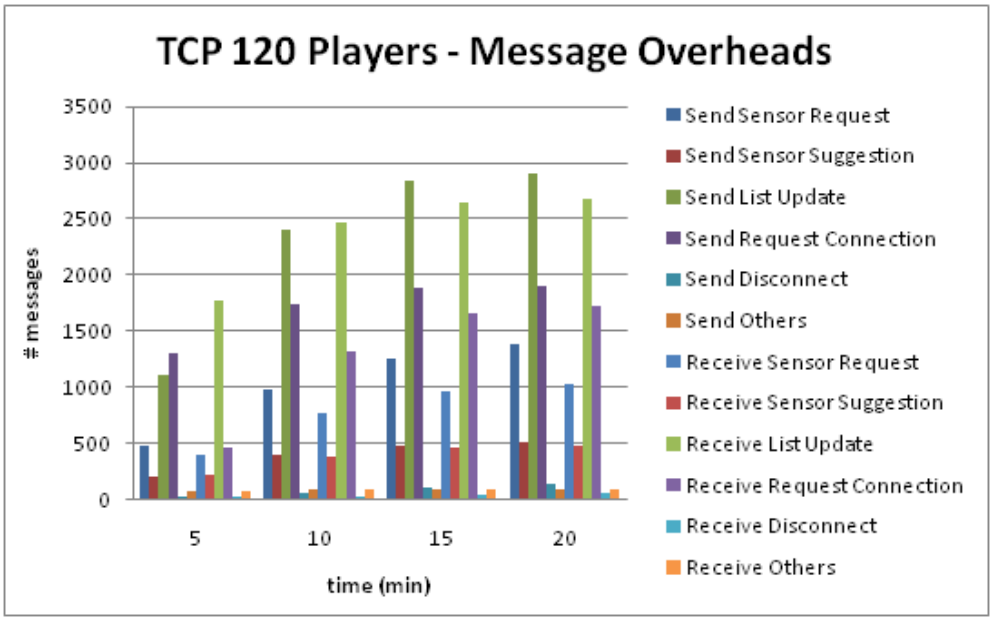


(b)

Figure 5.6: TCP Message Overheads

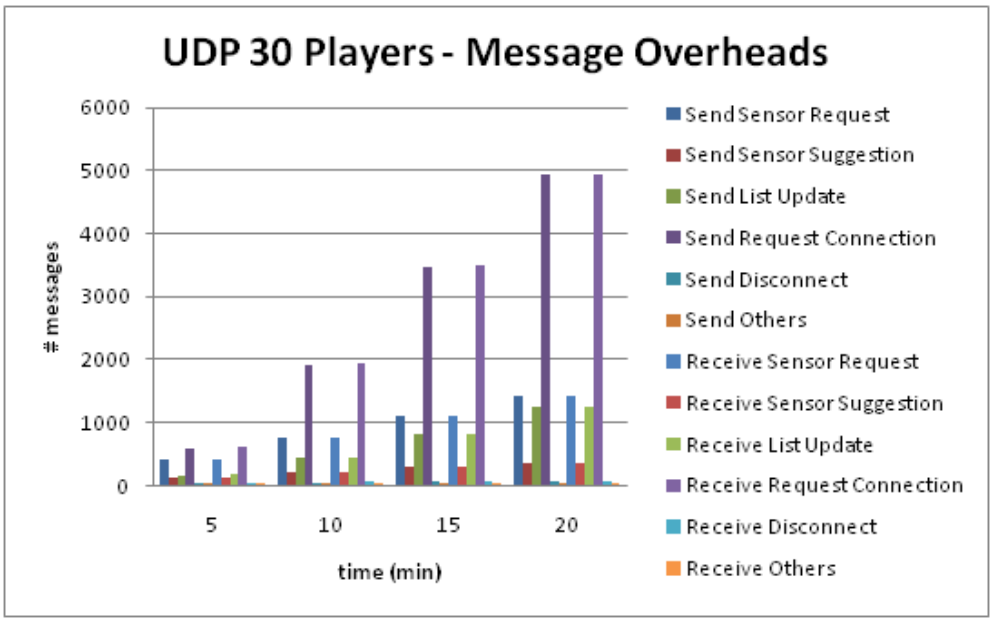


(c)

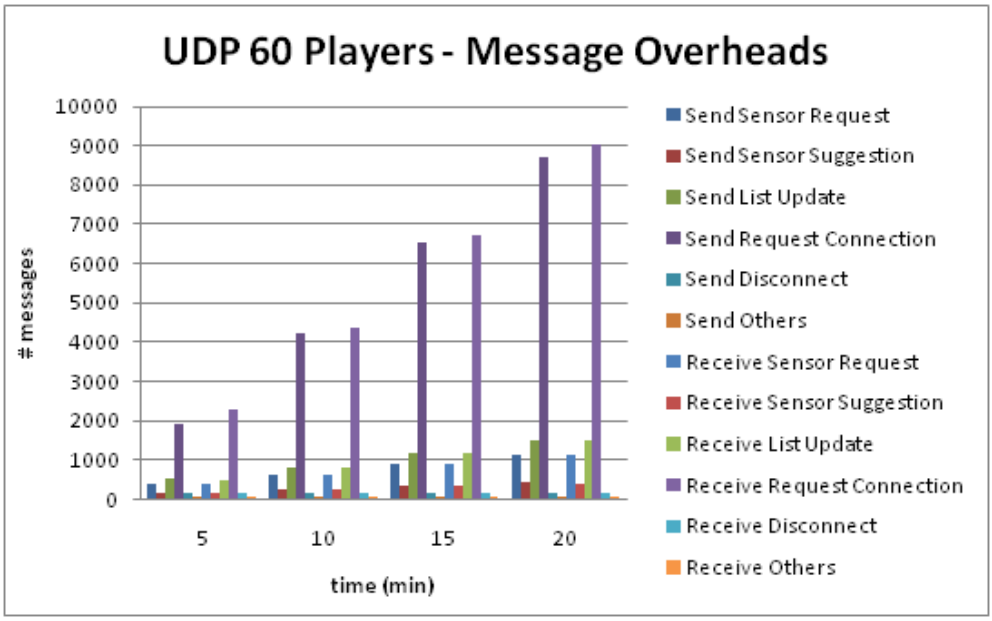


(d)

Figure 5.7: TCP Message Overheads (cont'd)

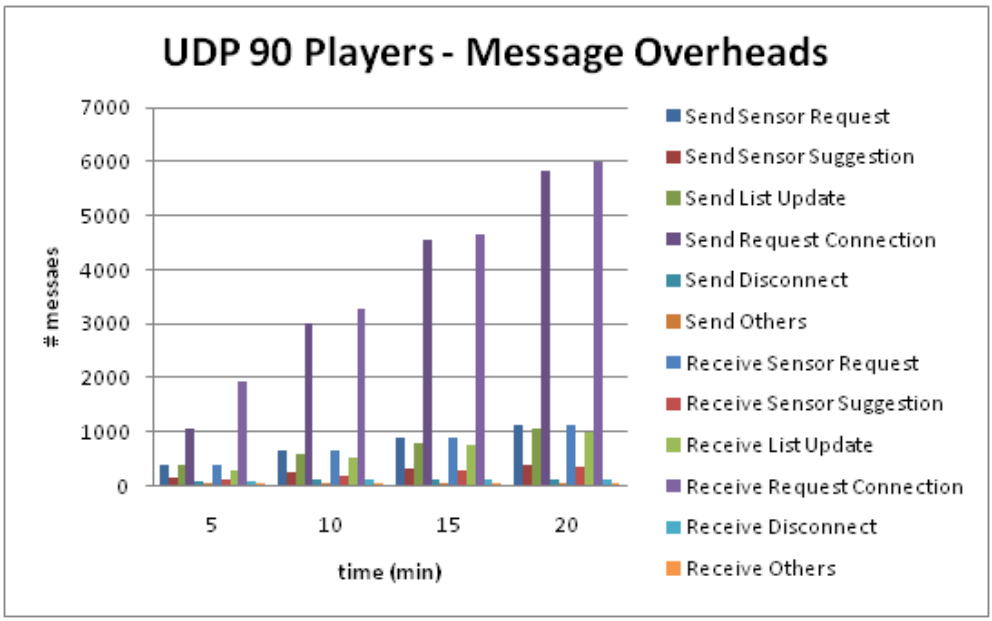


(e)

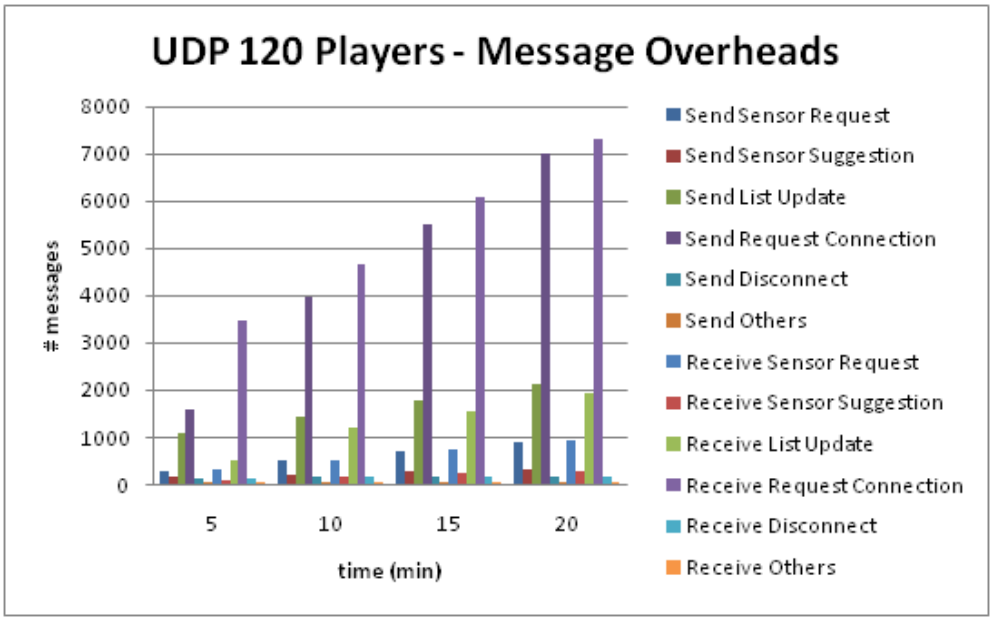


(f)

Figure 5.8: UDP Message Overheads



(g)



(h)

Figure 5.9: UDP Message Overheads (cont'd)

Chapter 6
Conclusions and Future Work

6.1 Conclusion

This thesis has shown that MMOGs can be run on a peer-to-peer network where player's position updates are multicast to interested peers. This overlay is dynamically built and updated by information gathered from peers. The performance of this network engine has been studied with two different transport protocols, which are TCP/IP and UDP/IP. Under normal load, TCP and UDP performance are very similar, despite the fact that more resources are consumed with TCP. Being resource greedy, TCP turned out to be less scalable as we were limited to a maximum of 3 NPC clients in one computer during the experiments. In a real game setting, each client machine will be running a single NPC client and thus that amount of resources used may not be as problematic. On the other hand, UDP scales up very well, but being an unreliable protocol, message losses can put a dent in its performance.

6.2 Future Work

There is a future for building massively multiplayer online games on a peer-to-peer structure using a localized multicast. However, there are limitations that need to be investigated or improved; implying a many potential future directions for this work.

6.2.1 Firewall

Our experiments were run on lab computers in the McGill's computer science network. Since they all belong to the same network, we did not have to account for situations where a firewall is in place or a client machine resides in a network using NAT. This means that most private users will not be able to connect to the peer-to-peer network and thus, will not be able to use a version of Mammoth with Yobol. From a home computer, when a client tries to connect to a node on a lab machine, the connection fails due to a firewall preventing the opening of a TCP connection to the client.

This can be solved by implementing the hole punching algorithm [FSK05] on top of the UDP implementation of Yobol. This would allow the traversal of firewalls and NAT networks.

6.2.2 Reliability

This research has shown that UDP is a preferable medium than TCP for MMOGs. UDP consumes much less resources, since no connection setup is needed before sending a message. Nevertheless, UDP is known to be an unreliable protocol. Since our experiments were run in a local area network, few message lost were observed, thus having less impact on the game. However as the load gets higher, messages are more frequently lost. This will eventually spoil the player's game experience. To remedy this, some of TCP's reliability can be implemented in UDP. A possible approach would be a local node sending the position update unreliably the first four times, and the fifth time it moves, the position is sent out in a reliable fashion. This ensures that peers will always receive some updates

6.2.3 Security

Being a peer-to-peer approach, security is a major issue since each client node controls a master object. Malicious nodes can send out erroneous data about their state in order to gain some unfair advantage on their peers, like making themselves invisible. They could also send out fake messages to overload the network and prevent it from working well. Since this topic has not yet been explored, further work is necessary.

6.2.4 pSense

Being inspired by pSense, we have focused on distributing player master objects on the client nodes and left other game objects on the server. This can be pushed further by distributing all mutable objects onto clients, making this a pure peer-to-peer approach. On the other hand, other interest management can be explored as well as other ways to allocate sensor nodes to optimize the performance.

Bibliography

[AG] 10Tacle Studios AG.
URL: <http://ng.neocron.com>.

[ARBS04] M. Agrawal A. R. Bharambe and S. Seshan. Mercury: supporting scalable multi-attribute range queries. In *SIGCOMM*, pages 353–366, 2004.

[Are] Inc. ArenaNet.
URL: <http://www.guildwars.com>.

[ASB08] S. Jeckel P. Kabus B. Kemme A. Schmieg, M. Stieler and A. Buchmann. psense - maintaining a dynamic localized peer-to-peer structure for position based multicast in games. In *Proceedings of the 2008 Eighth International Conference on Peer-to-Peer Computing*, pages 247–256, Washington, DC, USA, 2008. IEEE Computer Society.

[BK04] W. Xu B. Hopkins B. Knutsson, H. Lu. Peer-to-peer support for massively multiplayer games. In *INFOCOM*, pages 96–107, USA, 2004. IEEE.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[Ent] Blizzard Entertainment.
URL: <http://www.blizzard.com>.

[FSK05] Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-peer communication across network address translators. In *ATEC '05: Proceedings of the Annual Conference on USENIX Annual Technical Conference*, pages 13–13, Berkeley, CA, USA, 2005. USENIX Association.

[GCV01] I. Keidar G. Chockler and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[JK03] G. Simon J. Keller. Solipsis: A massively multi-participant virtual world. In *Proceedings of Parallel and Distributed Processing Techniques and Applications*, pages 262–268, 2003.

[JK09] B. Kemme A. Denault M. Hawker J. Kienzle, C. Verbrugge. Mammoth: a massively multiplayer game research framework. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, pages 308–315, Orlando, Florida, USA, 2009. ACM.

[JSBV06] J. Kienzle J.-S. Boulanger and C. Verbrugge. Comparing interest management algorithms for massively multiplayer games. In *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games*, page 6, Singapore, 2006. ACM.

[KPBM99] Ö. Özkasap Z. Xiao M. Budiu K. P. Birman, M. Hayden and Y. Minsky. Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2):41–88, 1999.

[LP96] J.C.-H. Lin and S. Paul. Rmtp: A reliable multicast transport protocol. In *INFOCOM*, pages 1414–1424, 1996.

[MCR02] A.-M. Kermarrec M. Castro, P. Druschel and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)*, 20(8):100–110, October 2002.

[min] Apache mina.
URL: <`http://mina.apache.org/`>.

[Mor96] K. L. Morse. Interest management in large-scale distribution simulations. Technical report, Department of Information and Computer Science, University of California, Irwine, 1996.

[Pla] Inc. Playnet.
URL: <http://www.battlegroundeurope.com>.

[PTEK03] S. B. Handurukande P. Kouznetsov P. T. Eugster, R. Guerraoui and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Tans. Comput. Syst.*, 21(4):341–374, 2003.

[RD01] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[YV05] A. P. Yu and S. T. Vuong. Mopar: a mobile peer-to-peer overlay architecture for interest management of massively multiplayer online games. In *NOSSDAV*, pages 99–104. ACM, 2005.

[Zin08] Dominik Zindel. Postina: A publish/subscribe middleware designed for massively multiplayer games. Master's thesis, McGill University, University of Fribourg, Montréal, Québec, Canada, Switzerland, 2007-2008.