Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing Group

Master Thesis

Towards Peer-to-Peer Games: XP2Pilot

Jean-Luc Geering
jlg@student.ethz.ch

Dept. of Computer Science
Swiss Federal Institute of Technology (ETH) Zurich
Summer 2007

Prof. Dr. Roger Wattenhofer
Distributed Computing Group

Advisors: Stefan Schmid, Thomas Locher

Abstract

A game is both an interesting and a complex application. Having a shared state makes it a challenging undertaking to develop it in a peer-to-peer setting. *XP2Pilot* is an implementation of such a game. It is a fully decentralized version of *XPilot-NG*, built on top of *Pulsar*.

Acknowledgments

First of all, I would like to thank Prof. Dr. Wattenhofer for giving me the opportunity to work on this most interesting and challenging project. During this truly unique experience, I benefitted not only on a technical but also on a personal level.

Stefan Schmid and Thomas Locher, my faithful advisors, thank you for not breaking my e-bike when you took it for a ride.

Special thanks to Remo Meier who allowed me to use *Pulsar* and was always quick to respond to my numerous questions.[1]

Cordial thanks to the open-source communities. They make the world a better place. At least the virtual one.

To the Swiss taxpayers, thank you for financing my education. Keep up the good work, the SFITZ[2] is such an awesome institution.

To the TIK, thank you for lending me Funghi. He was a good companion who never let me down during the six months we spent together. I wished all laptops would behave like he did.

To the Swiss Federal Railways, thank you for letting me feed Funghi on your trains. We could spend even more time working together.

Thanks to my family, friends and fellow students for playing my game when it was neither finished nor fun.

Thanks, Jean-Paul Würth. Your nonsense full of meaning helped me in a crucial moment.

To all those who pretended listening to me when I was complaining - I hope we can still be friends.

Let's not forget all the brave readers of this report:

Thank you and good luck!

[1] Yes, I know, I should have read the documentation first.
[2] Aka EPFZ aka ETHZ.

Contents

Chapter 1

Introduction

There are many peer-to-peer applications, but most of them do not have a shared state. On the other hand, games have to deal with player interaction, i.e. a shared state. Would it not be interesting to store and manipulate a shared state in a not only distributed but also fully decentralized system, in other words, build a peer-to-peer game?

1.1 Common Uses of Peer-to-Peer

1.1.1 File Sharing

The most prominent use of peer-to-peer technology nowadays is file sharing. The *Gnutella* network is one example of an early successful application, and today *Bittorrent* [1] is the dominant file sharing protocol.

The characteristics of file sharing are that the content is static, i.e. the file being shared does not change,[1] there are no timing constrains, and the goal is to transmit information from one to many other nodes. If we only look at the information that has been sent after the transfer is complete, it is equivalent to one server sending the same content to many clients.[2]

1.1.2 Multimedia Live Streaming

The new trend in peer-to-peer computing is to replace a streaming server with a peer-to-peer distribution scheme. In Switzerland *Zattoo*[3] is already a popular live video streaming system, although it does not deliver very good quality. *Joost* is a service by the makers of *Kazaa* and *Skype*. *Pulsar* [2], is a new peer-to-peer based protocol which promises good performance.

We find some of the same characteristics in streaming as in file sharing: the goal is to distribute content from one server to many clients in an efficient and robust way. If we consider each frame of a video stream separately, the content is also static, but small and with a short life-span. But contrarily to file sharing there are timing constrains. The frames have to be delivered in order and on time. A delay of a few seconds is acceptable, but certainly a few hours or days like we know it from file sharing applications, is out of question.

[1] In case it does change, it will be shared as a new file.
[2] Of course it is far more efficient, and even if the first node goes offline the data keeps spreading.
[3] http://www.zattoo.com

1.1.3 Distributed Hash Tables

A distributed hash table (DHT) is a fully decentralized system for storing, looking up and retrieving key-value pairs. Important features include scaling to a large number of nodes, fast message routing and good performance and resilience under churn. Well known examples of DHTs are CAN [3], Chord [4] and Pastry [5].

A DHT consists of a keyspace, i.e. the set of allowed keys, for example the 16 bytes bit strings, and a keyspace partitioning scheme describing how to split ownership among participating nodes.

To illustrate this, let us consider a circular keyspace. Each node has its own ID that serves as a key or, if it is not in the right format, can be hashed into a key. This ID defines its position on the circle. The node owns and is responsible for all the keys between its ID and the ID of the next node on the circle.

The nodes connect to neighbours and arrange themselves in an overlay network. The different protocols define to which and to how many neighbours a node has to connect. This determines how fast a protocol can route messages, how well it scales and handles churn.

The data stored in a DHT is also static. Direct modification of the values is not possible. To update a value, a new value is inserted with an identical key, to replace the old one.

1.2 Peer-to-Peer and Games

Games are an interesting application for peer-to-peer computing. While a shared state and player interactions need to be taken care of (i.e., the content is not static), absolute correctness and consistency do not have to be guaranteed (a game is no banking software). As long as they do not trouble the user too often, errors can be tolerated.

In order for a game to be easily distributed, locality of interest is essential [6]. A peer-to-peer game where every node needs to know about everything is certainly feasible, but it would not benefit of most advantages of a distributed system. Depending on the game, a node's visibility can be limited without influencing the user experience, e.g. any 2D war game with limited screen size and fog of war.[4] On the contrary, it would be absurd to play chess seeing only a quarter of the board at the time.

There are also timing constrains to be considered when implementing a game, as the user expects a fast feedback for his actions. In this sense, a game might be even more sensitive to delays than a live streaming application.

Any game will have its share of cheaters, and so will peer-to-peer ones. The architecture of the system should include anti-cheating mechanisms whenever possible, and make it hard to cheat when it cannot be prevented.

[4] [7] further reduced the amount of data transmited by defining *focus sets*.

Chapter 2

Components

2.1 Game

Instead of writing a new game from scratch, we chose a existing open-source client-server game. Two multiplayer games, both realtime, were considered:

Quake 3 is a first person shooter and has been used many times for research (e.g., [7], [8]). Its engine is freely available, but some components still have to be bought to actually run the game. The game is also fairly complex, has advanced graphics and dead reckoning.

XPilot (*Fig. 2.1*) is an *Asteroids*-like 2D multiplayer game, where items can be collected to enhance the offense or defense characteristics of the player's ship. It is the game used by [9] to implement their model. It is open-source,[1] and rather simple. It can be simplified further by removing features like items, shields and fuel.

The main reason to choose *XPilot* was it's simplicity compared to *Quake 3*, making it easier to adapt, as the focus was more on peer-to-peer aspects than on a game with many features or nice graphics.

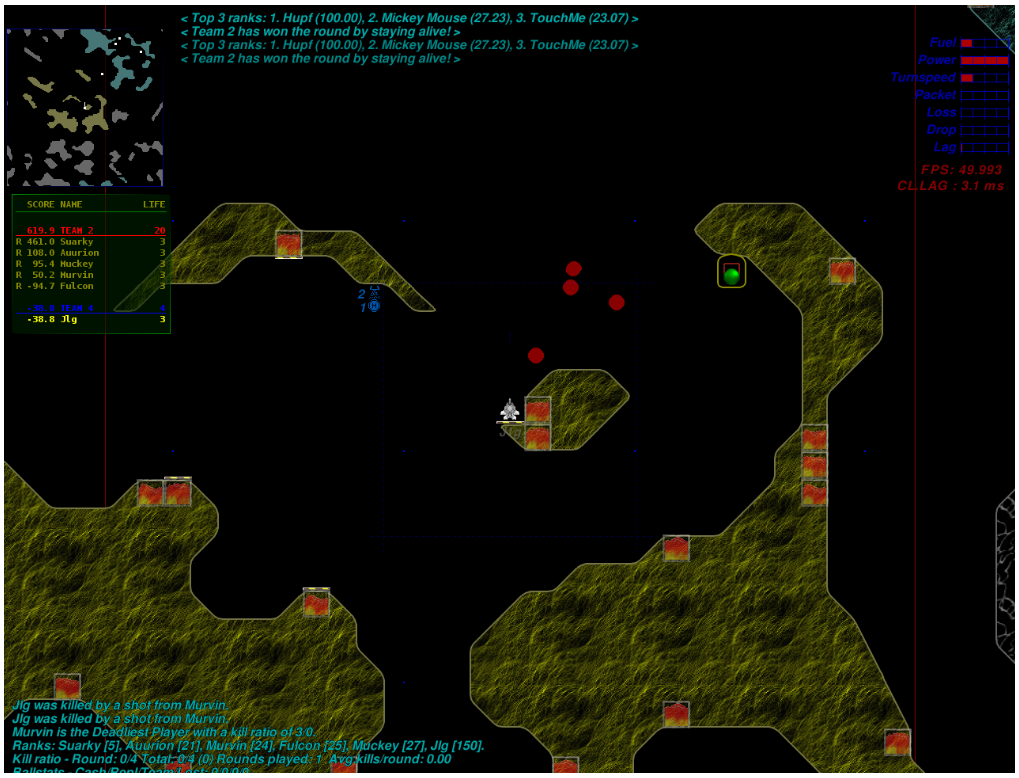
2.2 Network: Peer-to-Peer Overlay

As the goal was not to develop yet another type of peer-to-peer overlay network, we decided to reuse an existing DHT. We only needed a protocol with an available implementation, which could build and maintain an overlay network and route messages in it in a relatively short time. We tried to keep our game implementation as independent as possible from the underlying protocol and its implementation.

The first versions of our prototype were running on *FreePastry*[2], a free java implementation of the *Pastry* [5] protocol. The main drawback was that a node cannot signal that it wants to leave the network. Its absence is only detected after a timeout. But for our game to be responsive, we wanted to avoid as many timeouts as possible.

Therefore, we switched to *Pulsar*[3], which provides all required features, in particular fast message routing and keyspace management.

[1]Available from sourceforge.net.
[2]Available at http://freepastry.rice.edu/.
[3]With the friendly permission of its creator.



```
< Top 3 ranks: 1. Hupf (100.00), 2. Mickey Mouse (27.23), 3. TouchMe (23.07) >
< Team 2 has won the round by staying alive! >
< Top 3 ranks: 1. Hupf (100.00), 2. Mickey Mouse (27.23), 3. TouchMe (23.07) >
< Team 2 has won the round by staying alive! >
```

```
  SCORE NAME         LIFE

  619.9 TEAM 2         20
R 461.0 Suarky          3
R 108.0 Auurion         3
R  95.4 Muckey          3
R  50.2 Murvin          3
R -94.7 Fulcon          3

  -38.8 TEAM 4          4
  -38.8 Jlg             3
```

```
FPS: 49.993
CLLAG : 3.1 ms
```

```
Jlg was killed by a shot from Murvin.
Jlg was killed by a shot from Murvin.
Murvin is the Deadliest Player with a kill ratio of 3/0.
Ranks: Suarky [5], Auurion [21], Murvin [24], Fulcon [25], Muckey [27], Jlg [150].
Kill ratio - Round: 0/4 Total: 0/4 (0) Rounds played: 1  Avg.kills/round: 0.00
Ballstats - Cash/Repl/Team/Lost: 0/0/0/0
```

Figure 2.1: *xpilot-ng-sdl* screenshot.

Chapter 3

Architecture and Design

XP2Pilot is a fully decentralized game. In this chapter we will describe the game, the architecture of its components and how it integrates *XPilot-NG* and *Pulsar*. We will also see the three roles a node plays, and its principal data structures.

3.1 The Game

XP2Pilot is a simplified version of *XPilot-NG*. The user controls a ship which has to be navigated around obstacles (walls), looking for the ships of other players. He[1] is supported in this task by a short range radar indicating the direction and distance of surrounding ships. To determine his position at any moment during a game, the user can look at a map indicating the current position of his ship in the world.

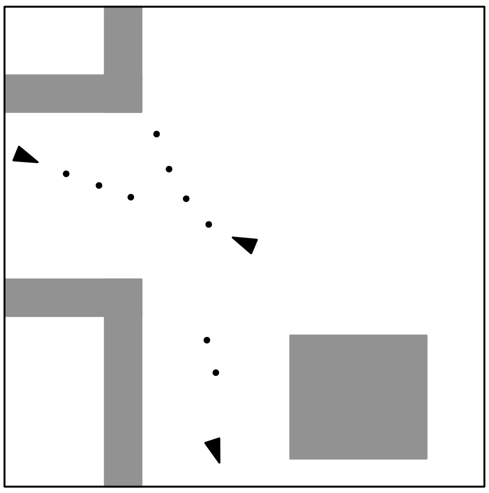


Figure 3.1: Schematic screenshot of the game, showing its 3 main elements: triangles represent ships, dots are bullets fired by a ship and the grey zones are walls. The ship in the center is always the one controlled by the player.

Once he comes close to another ship, he can fire bullets and try to destroy it. Of course, his opponents will do the same, quickly filling the space with bullets. In order not to

[1]Or she. Out of sheer conventionalism or laziness, we have opted for the male form throughout this thesis - which does not mean that female users could not appreciate the game.

lose points, the user has to avoid being hit by a bullet or crashing into another ship or into a wall while chasing his adversaries.

Users lose one point every time their ship is destroyed by an adversary and 2 points if the destruction is self inflicted (e.g., by crashing into a wall or flying into one of their own bullets).

The difficulty of the game resides in the fact that the player has limited control over his ship. He can only change its direction, its speed and fire bullets along the axis of the ship. To slow down, he has to turn his ship 180° around and accelerate. As the cannon is mounted in front of the ship and always fires straight ahead, it takes a bit of practice to maneuver the ship while aiming at a moving target.

3.2 From ***XPilot-NG*** to ***XP2Pilot***

XPilot-NG is divided in two parts, a client and a server (*Fig. 3.2*), which communicate over UDP. In order to build *XP2Pilot* as a peer-to-peer system, we discarded the server and kept only the client[2], which was used as a GUI for the game. It communicates with a pseudo server written in java, using the original xpilot protocol (*Fig. 3.3*). This server is only a facade in front of the game engine. It formats the data for the client and sends it using the xpilot protocol. Furthermore, it decodes the commands sent back by the client and passes them to the game engine.

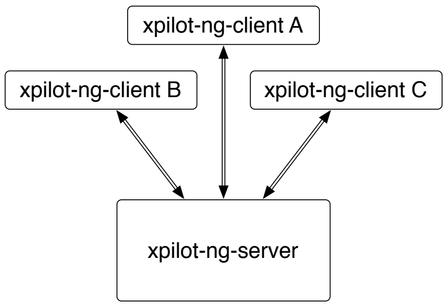


Figure 3.2: Client-Server architecture of *XPilot-NG* with 3 players.

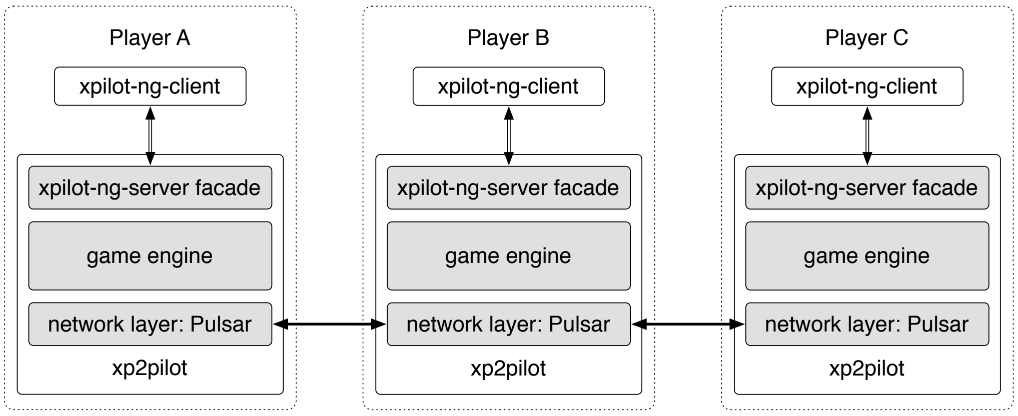


Figure 3.3: P2P architecture of *XP2Pilot* with 3 players.

[2]In Ubuntu Linux it corresponds to the `xpilot-ng-client` package.

3.3 Network Layer

Pulsar provides an abstraction of the network layer. It communicates with other instances by multiplexing messages over one UDP socket and generates a unique identifier from the IP address and port number of this socket. This identifier has different functions: it is the player's ID, is used for routing and determines what responsibilities a node has. In addition to the messages needed for building and maintaining the overlay network, *Pulsar* transmits messages for the game engine. It also signals when a new node joins, leaves or times out.[3]

Each node is responsible for all the IDs closest to its own ID. A node can decide if an ID is local by comparing it with its own and with the IDs of all its neighbours.

3.4 Game Engine

The game engine (*Fig. 3.3*) is the core of the system and contains the actual logic. It has a small physics engine, manages the state of the game and scores of the players, collaborates with other nodes and adapts to any change in the overlay network. It also caches all the data located in other nodes that is needed by the client.

These tasks can be divided into three groups according to the various roles a node plays:

3.4.1 Server Role

XP2Pilot is played in a 2-dimensional square world[4], which is divided into smaller squares. For convenience, we shall call each of these small squares a `piece`[5]. Each `piece` has a key (an ID) and the keys are evenly distributed in the keyspace of the underlying DHT, i.e. *Pulsar*. Using *Pulsar*'s keyspace partitioning scheme, the responsibility for a `piece` is assigned to a node, which acts as a server for this `piece`.

To guarantee some continuity in case of a server node failure, each one of them sends a periodical update to one or more backup nodes. These nodes are also defined by the mechanism which is used to find a server node, the first backup node being the node that will own the given `piece`'s ID when the current server node leaves.

For each `piece` it is responsible for, a node keeps a list of the ships and the bullets currently located in it, and a list of the nodes subscribed to it (*see 3.4.3*). At every time-step, a series of tasks have to be performed:

- Apply the latest command sent by a user to his ship.
- Create new bullets for the ships that are firing.
- Update the position of all objects (ships and bullets).
  - Bullets hitting a wall are removed.
  - Bullets whose time to live has expired are removed.
  - Ships colliding with a wall are either reflected or destroyed if their collision speed exceeds a given speed limit.

[3]More precisely: it notifies the game engine when connecting to a neighbour, or when losing this connection.

[4]A 2D torus to be more specific, as the left side is connected to the right and the top to the bottom.

[5]In the code, they are implemented by the `WorldPiece` class, not to be mistaken for `WorldPeace`!

  - Ships colliding with a bullet or with each other are also removed.

- Notify the registrar (*see 3.4.2*) node of each removed ship that it has been destroyed by a bullet of a given player. A ship crashing into a wall is treated like a ship destroyed by one of its own bullets.
- Transmit the objects moving out of the `piece`'s boundary to the node responsible for the adjacent `piece`.
- Send the continually updated game state, i.e. the list of the positions of all objects, to all subscribed nodes.

3.4.2 Registrar Role

When a player connects, he registers his ID with the node closest to the inverse of his own.[6] Conversely every player is a registrar for others. Every node keeps a registry, which contains ID, nickname, and score of the registered players.

The registrar node is also responsible for the ship of the player node. When the player's ship gets destroyed, either by getting hit by a bullet, by colliding with another ship, or crashing into a wall, the registrar is notified by the server node. The registrar updates the score and inserts the ship at a random new position.

The registrar node also tries to make sure that there is always one and only one copy of each of its players' ships. If needed, it can insert a new ship or remove a duplicate.

3.4.3 Client Role

The player or client role consists in gathering and keeping up to date the information presented to the user, i.e. prepare the data for the *xpilot-ng-server facade* layer.

The client node must keep track of its user's current ship position, which server node corresponds to this `piece`, and forward the user's commands to this node. As will be seen in *3.5*, a node can be in the client and the server role at the same time for a given `piece`.

A client node only needs to know what is happening in the area around its user's ship. Thus, the node subscribes to the server node of the visited `piece` and to those of the eight `pieces` around. It will only receive updates for these nine `pieces`. This active zone follows the player's ship. It is shifted when the ship crosses a `piece` boundary.

When coming across a ship with an unknown ID, the node queries the registrar of this ID for the nickname and the score of the player controlling the ship. It keeps polling the registrar at regular intervals to maintain its local score list up-to date. (*Fig. 3.10.b*)

3.5 The Pieces and their State

A node can be the server for a subset of `pieces`[7] and have another subset in the active zone. To manage this, each `piece` is flagged when it is active (i.e., in the active zone) and in a given state. There are three main states: *Void*, *Remote* and *Local*, and four transition states between them (*Fig. 3.5*).

[6]For more than 1 node this is always a remote node.

[7]For a definition of `piece`, see the first paragraph of *3.4.1*.

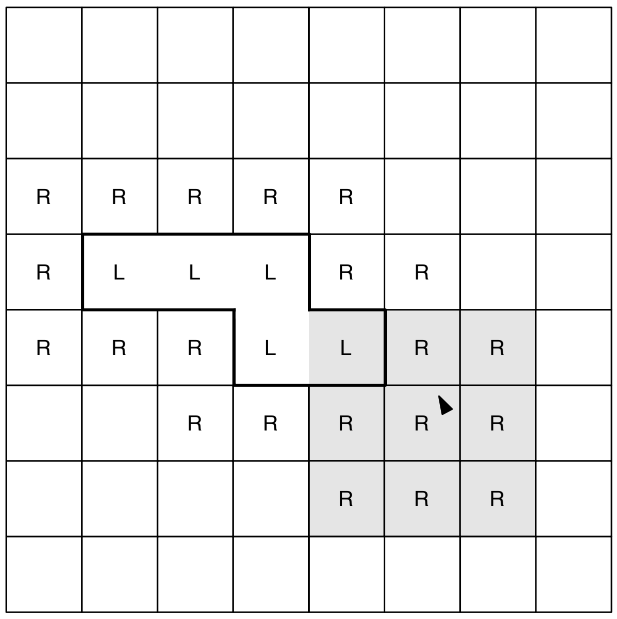


Figure 3.4: The world as seen by a node. The little triangle represents the player's ship, they grey zone around it represents its active zone. The letters in the `pieces` (i.e., the little squares), if present, indicate their state: L stands for *Local*, R for *Remote*. `Pieces` without letters are in the *Void* state.

- A `piece` is in the *Local* state if the node owns its ID, i.e., when the node is the server for the `piece`.
- A `piece` is in the *Remote* state when the node is subscribed to the server node in charge of this `piece`. This can be for two reasons:
  - If it has the active flag set[8] but is not *Local*.[9] The node needs continuous information about the situation in this `piece` in order to display it to the user. This data is contained in the remote updates sent by the server.
  - If one of the adjacent `pieces` is in the *Local* state. In this case the node needs to know the address of the server node in order to be able to transfer objects without delays.
- The `pieces` in the *Void* state are of no current interest for the node.
- *Subscribe*, *Unsubscribe*, *Transfer_in*, *Transfer_out* are the transition states between the 3 previous ones (*Fig 3.5*). They will be described in more detail in the next section.

3.6 Network Dynamics

As users join and leave the game, nodes have to constantly adapt to their new neighbourhood. A new node modifies the partition of the keyspace and thus the ownership of `pieces` and the re-partition of `players` between the registrars. The same happens when a user quits.

[8]i.e., it is one of the nine `pieces` in the active zone.
[9]If the `piece` is *Local* the information is already available and there is no need for subscription.

When a node leaves gracefully, it sends the state of its local `pieces` and the content of its registry to their respective nearest neighbours, notifies all of its subscribers, and signals the underlying network layer it should leave the overlay network.

When a node fails, its absence will be detected by its neighbours only when a timeout happens. The data has to be recovered from existing backups or is lost if none is available.

As ships move around the world they cross `piece` boundaries, thus modifying the active zone. This can cause a change of state for the `pieces` that are not any more in the zone, and new `pieces` may have to be subscribed to.

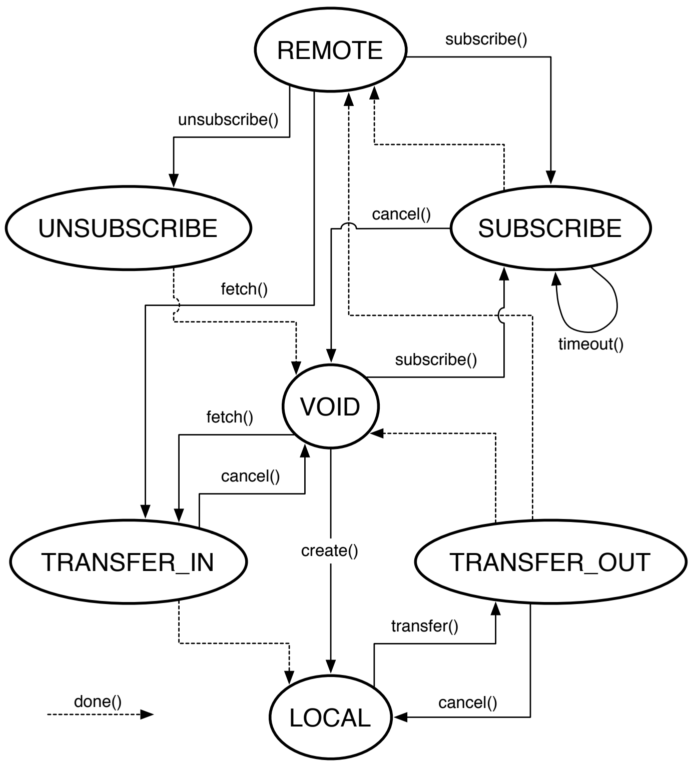


Figure 3.5: State transition graph for a `piece`.

3.6.1 Piece Subscriptions

When a node detects that it needs to subscribe to a `piece`, it sets the `piece`'s state to *Subscribe* and sends a *piece subscription request* message to the server node which is in charge of this `piece`. If all goes well it gets a *piece subscription ack* message back from the server, allowing it to switch the `piece`'s state to *Remote*. If it does not get this message back, a timeout (*Fig. 3.5*) will trigger the sending of a new *piece subscription request* message.

On receiving a *piece subscription request* message, the server node adds the subscriber to the subscriber list for the given `piece`. After replying with a *piece subscription ack* message, it starts sending regular *remote update* messages containing the status of all objects currently in the `piece`.

Fig. *3.6* illustrates this process. On the left side, it shows the `piece` state for the subscriber node, on the right side for the server node, and in between the messages they exchange. As noted in section *3.5* a node never subscribes to a *Local* `piece`.

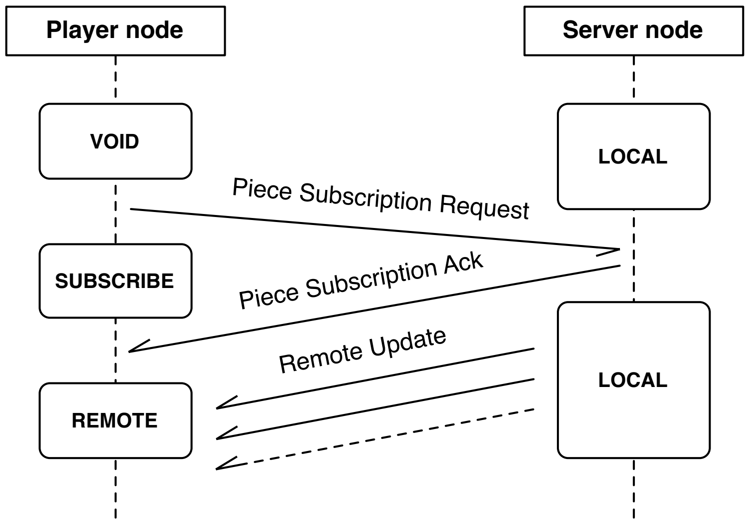


Figure 3.6: Subscription process. Illustration of the state of the same `piece` on the subscriber and on the server side. Messages are sent asynchronously.

Unsubscribing is even simpler as the node does not have to wait for an acknowledgment. It will detect that the server node did not receive its message if it keeps receiving *remote update* messages.

3.6.2 Piece Transfer States

The actual transfer between the current server node and the new one happens in three phases. When the new node determines that it is responsible for a `piece`, it places it in the *Transfer_in* state (*Fig. 3.5*) and sends a *piece transfer request* message, containing the `piece` number, to the current server. (*Fig. 3.7*)

Upon receiving this message, the old server node changes the state of the `piece` from *Local* to *Transfer_out* (*Fig. 3.5 & 3.7*). It replies with a *piece transfer data* message containing all the ships and bullets currently in this `piece`, a list of the nodes subscribed to it, and a flag indicating if itself wants to subscribe to this `piece`. It also notifies all the subscribers which new node is taking over the responsibility as a server with a *piece subscription transfer* message. (*Fig. 3.8*)

When it receives the data, the new server node sets the state of the `piece` to *Local*, answers with a *piece transfer ack* message (*Fig. 3.7*), and sends a *piece subscription ack* message to all the subscribing nodes of the incoming *piece transfer data* message (*Fig. 3.8*). If requested, the old server node is added to the list of subscribers for the `piece`.

The *piece transfer ack* message lets the old server set the state of the `piece` to *Void* like in *Fig. 3.7*, or if requested the subscription, directly to *Remote* (*Fig. 3.5*).

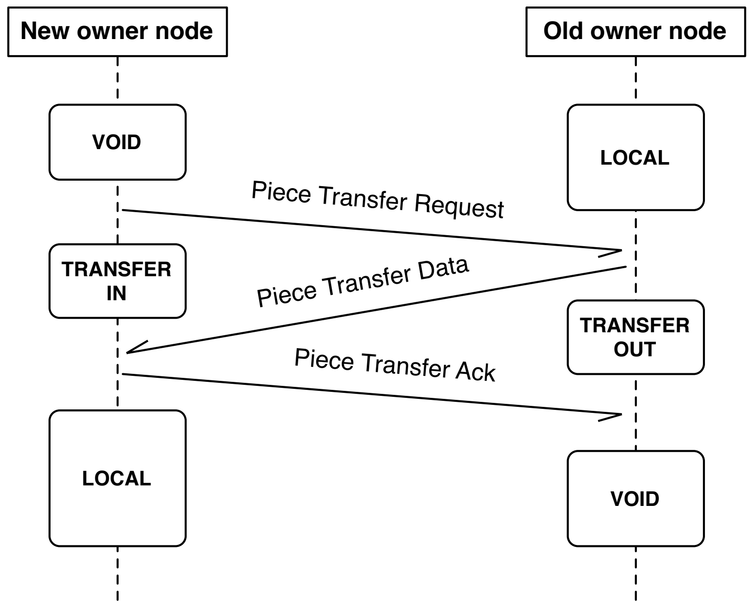


Figure 3.7: `Piece` transfer process. For the considered `piece`, it shows the state transitions in both nodes. Messages are sent asynchronously.

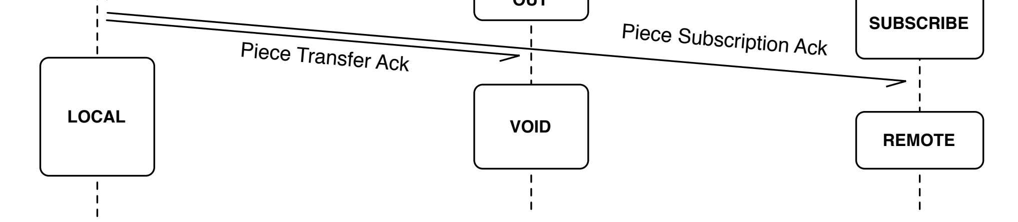


Figure 3.8: Combined `piece` transfer and subscription transfer process involving three distinct nodes. Messages are sent asynchronously.

3.7 Network States

A node can be in four distinct network states:
Connect, *Join*, *Ready* and *Leave*. (*Fig. 3.9*)

- It starts in the *Connect* state. In this state, it looks for nodes to connect to. It can use multicast messages on the local area network or contact a pre-configured *Pulsar join server* and ask it for a list of recently active nodes.

- As soon as it finds a node, it switches to the *Join* state and gives *Pulsar* some time to initialize the overlay network. It starts sending notifications[10] of its state to its neighbours. It receives the same from them together with *piece location hint* messages which it then forwards to all of its neighbours also in the *Join* state. These messages help the node to find the current server node of a `piece` in case multiple nodes join simultaneously.[11]

- After a given time, the node is considered to have reached its position in the overlay network and the state is set to *Ready*. The node requests the `pieces` it is responsible for (*see 3.6.1*) and starts answering registration requests from player nodes. It registers its user and the actual game starts as soon as it gets an answer from the registrar.[12]

- When the state is set to *Leave*, the node quickly transmits its local `pieces` and the content of its registry, and signals *Pulsar* to disconnect from the neighbour nodes.

It could happen that a node loses all its neighbours when in the *Join* or *Ready* state. In this case, it simply goes back to the connect state and starts looking again for new neighbours.

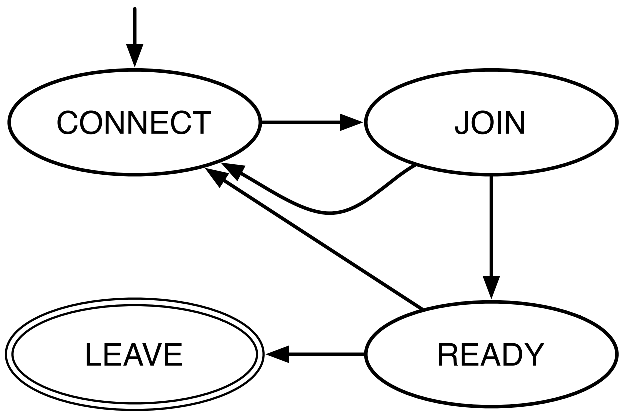


Figure 3.9: Network state transition graph.

[10] *GameNotification* messages.

[11] Let us consider the case where node A and node B are joining the game, and node S is the server for a given `piece`. Let us choose that A's ID is the closest one to the `piece`'s ID and that B's ID is closer to it than the ID of S. Without *piece location hints*, A would request the `piece` from B, which does not have it, and B would not request it from S knowing that A owns the `piece`'s ID.

[12] This answer includes the position of the player's ship, for the node might need to subscribe to the `piece` where the ship is located.

3.8 Player Management

When a player node joins the game, it sends a *Register* message containing the nickname of its user to its registrar, i.e. the node responsible for the inverse of its own ID.

The registrar creates a new entry in its registry with player ID, nickname, score, current_piece, and life_count[13]. It initializes the life_count to 1, and picks a random `piece` to be the current_piece. It then replies with a *current piece* message, indicating where the player can expect to find its ship, and a *score* message confirming that the nickname was registered[14] (*Fig. 3.10.a*). At the same time, the registrar node sends an *insert ship* message to the server node of to chosen current_piece (*Fig. 3.10.c*). This message contains the current_piece number, the player's ID and the life_count.

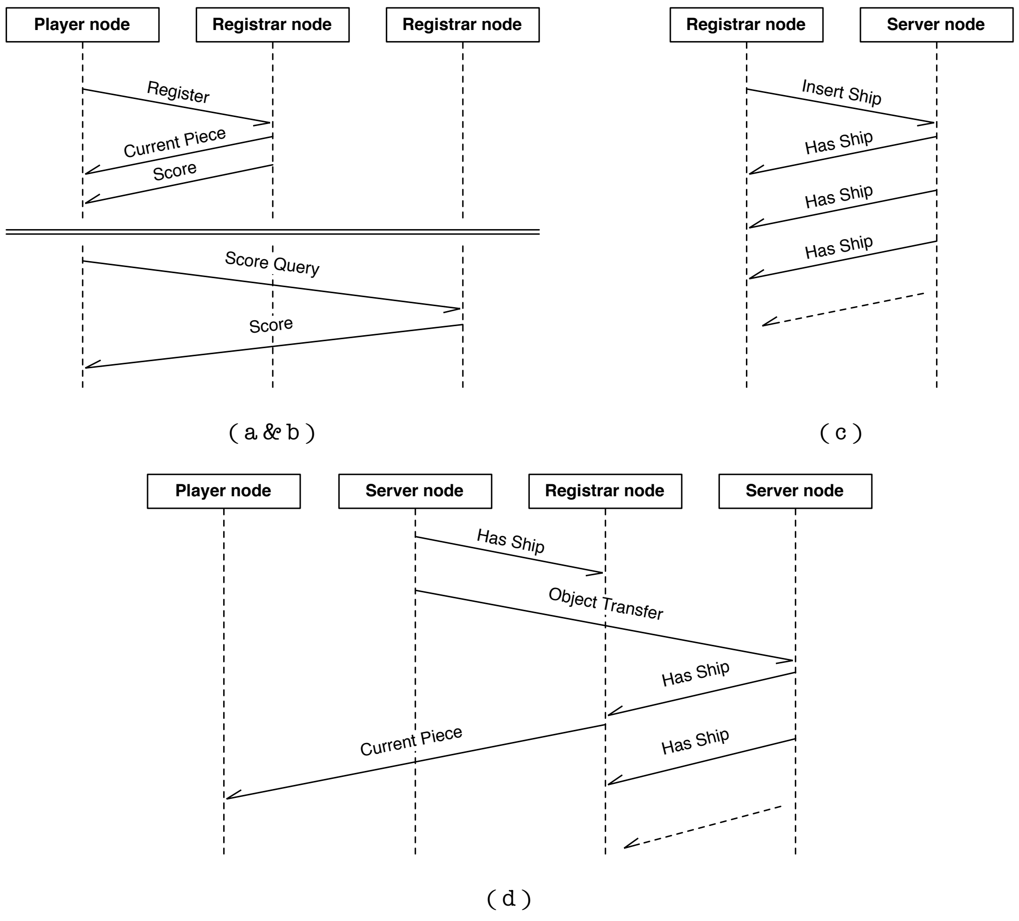


Figure 3.10: (a) Registration process. (b) Querying for the score and nickname of a user. (c) Inserting a new ship. (d) Ship transfer between 2 server nodes.

The server node inserts the ship in the `piece` chosen by the registrar. The ship is then managed like all other objects in this `piece`. The server node periodically sends back a *has ship* message containing the player's ID, the `piece` number and the life_count.

[13]Corresponds to a ship sequence number.
[14]The *score* message contains player ID, nickname and score.

On getting a *has ship* message, the registrar first compares the life_count with the stored life_count of this player. If they do not match[15], it replies with an *erase ship* message specifying the player's ID and the life_count. If the life_counts do match, the registrar updates the current_piece and notifies the client node (*Fig. 3.10.d*). This current_piece updating happens every time a ship crosses a `piece` boundary.

Object transfer messages (*Fig. 3.10.d*) are used when a ship or bullet crosses a `piece` boundary and needs to be transported to another server node.[16] When the object is a ship, the registrar learns about the transfer either by getting a *has ship* message from a new node (*Fig. 3.10.d*), or when reading a updated current_piece number in the *has ship* message of the previous server node.

When a ship is destroyed (*Fig. 3.11*), the server node notifies the registrar node with a *ship destroyed* message containing the player's ID and the life_count. If the life_counts match, the registrar updates the score, increments the life_count, picks a random `piece` as the new current_piece and initiates the ship insertion process as if the ship was the one of a player who just started.

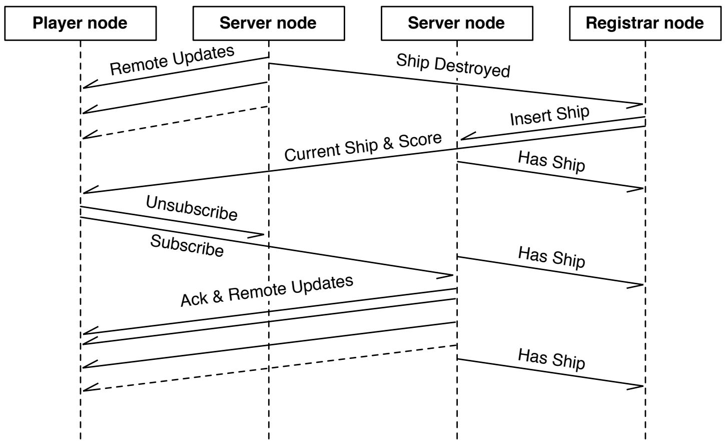


Figure 3.11: Replacement of a ship after it has been destroyed. To simplify the diagram, a *current ship* message and a *score message* are represented as one. The same holds for the *ack* and *remote update* messages. The player node only sends the *unsubscribe* message to the first server node if necessary (see *3.5*).

For better readability, the *command* messages[17] sent by the player nodes to the server nodes have been omitted in all of the diagrams.

3.9 A Final Comment

Throughout this chapter, we depicted the player, server and registrar as three distinct nodes for the sake of clearness. Nevertheless, it might happen that for a given scenario, a node has the player and the server role or the server and the registrar role. The only guarantee is that a player node cannot be its own registrar.

[15]Which can happen when the registrar decides to insert a new ship, wrongly assuming it has been lost although only the *has ship* message has been lost or delayed.

[16]When an object passes from a *Local* to a *Local* node, no special message is sent.

[17]See *3.4.3*, second paragraph.

Chapter 4

Implementation

XP2Pilot requires Java 1.5 and has 4 dependencies: *Pulsar* (pulsar.jar), *Spring* (spring.jar and commons-logging.jar) and *Log4j* (log4j-1.2.14.jar).
All classes are in the `ch.dongxi.xpilot`[1] package and its sub-packages.

4.1 Spring

The *Spring* application framework was used extensively throughout this project as a dependency injection container. Most important classes were designed as *Spring* beans. This allowed for a great flexibility when deploying launch configuration and simplified the reuse of components.

The *Spring IDE*, an *Eclipse* plugin, allowed an easy navigation of the project and its graph generation feature provided a good overview. (*see Appendix A*)

The *Spring* configuration files are included in the classpath, in the `springconfig` folder. The `real` subfolder contains configuration files that are specific to run *XP2Pilot* in a real environment while files in the `sim` subfolder are their counterparts for a simulation.

4.2 The `ch.dongxi.xpilot.game.*` classes

In the `Game` class we store the `NetworkState` (*Fig. 3.9*), and the `PlayerState` (i.e., `UNREGISTERED`, `REGISTERING` and `REGISTERED`).

The `Player` class is used by a registrar node to store a player's registration (*see 3.8*), and by a client node to store the score and nickname of other users. (*see 3.4.3*)

The `ShutdownHook` class adds a shutdow hook to the virtual machine, i.e. a thread that will be started when the user quits the game. (*see description of Leave in 3.7*)

4.3 The `ch.dongxi.xpilot.legacy.*` classes

All classes in the `ch.dongxi.xpilot.legacy` package and its sub-packages relate to the *xpilot-ng-server facade* layer. (*Fig. 3.3*)

[1]dongxi.ch is the author's personal domain.

The `ch.dongxi.xpilot.legacy.ConnectionSetupTask` class opens a UDP socket and listens for an incoming connection from the *xpilot-ng* client. The xpilot protocol multiplexes a reliable connection over this UDP connection. The reliable connection is used to transmit the map data and information about users joining and leaving. The UDP connection is used to send frames and receive keyboard and mouse commands from the user.

The `ch.dongxi.xpilot.legacy.SendFrameTask` class uses the `ch.dongxi.xpilot.legacy.net.UDPConnection` set up by the `ConnectionSetupTask` to send frames to the client at a frequency defined by the `FPS` constant.

The `ch.dongxi.xpilot.legacy.SendScoreTask` class uses the `ch.dongxi.xpilot.legacy.net.ReliableConnection` set up by the `ConnectionSetupTask` to keep the internal `ch.dongxi.xpilot.game.PlayerList` in sync with the one displayed to the user.

The `ch.dongxi.xpilot.legacy.XPilotClient` class spawns a new process starting the *xpilot-ng* client using the command specified in the options. It appends the `-join` and the `-port`[2] option to the command line. It redirects the output from `stdout` and `stderr` to the logging system (*log4j*).

4.3.1 The `ch.dongxi.xpilot.legacy.map` package

The `ch.dongxi.xpilot.legacy.map.LegacyMap` class reads a *XPilot-NG* map from a file in the *xp2* format (xml). The filename can be set with the `FILENAME` constant.

It provides the list of walls as a collection for the game engine and a *MapData* packet for the *xpilot-ng-server facade* layer.

4.3.2 The `ch.dongxi.xpilot.legacy.net.packet` package

The `ch.dongxi.xpilot.legacy.net.packet` package contains a class for most of the packet types defined by the xpilot protocol, and a `PacketFactory` class, used to read a `ch.dongxi.xpilot.legacy.Packet` from a byte stream. `ch.dongxi.xpilot.legacy.Packet.Type` is an `enum` listing all the packet types and their corresponding byte value.

4.4 The `ch.dongxi.xpilot.net.*` classes

The `ch.dongxi.xpilot.net` package contains a `ByteReader` and a `ByteWriter` to read from respectively write to a byte array. They can handle `boolean` values, `int` and `long` values using 1, 2, 4 or 8 bytes and `Strings`.

4.5 The `ch.dongxi.xpilot.p2p.*` classes

The classes in the `ch.dongxi.xpilot.p2p` package provide most of the game's logic.

The `CommandManager` forwards the user's command to the `CommandManager` of the right node from where it will be applied to the right ship. (*see 3.4.3 and 3.4.1*)

The `ObjectManager` implements the physics engine: it updates the position of ships and bullets, and checks if they collide with walls or with each other. It inserts or removes

[2] the port number is read from the `ConnectionSetupTask`

ships when told so by the registrar, and tells it about ships that have been destroyed (*see 3.8*). It handles object transfers when ships or bullets cross piece boundaries.

The `PieceManager` handles the pieces' state. It takes care of the necessary piece transfers when the overlay network changes (*see 3.6*), and subscribes to piece's server node when needed (*see 3.5*). Constants determine the intervals or timeouts for various maintenance jobs, for example `BACKUP_INTERVAL` is the interval between sending two piece backups.

The `PlayerManager` implements parts of the player role. It registers the user, and periodically queries registrars for the score of known users. (*see 3.4.3 and Fig. 3.10.b*)

The `PlayerRegistrar` class is responsible for the registrar role of a node (*see 3.8*). It keeps a registry of players and handles score queries. It manages each player's ship (intervals and timeouts are also determined by constants) and transfers a player's registration when a change in the overlay network requires it. (*see 3.6*)

The `*DTO`[3] classes are used to send objects, players, pieces and remote updates between nodes.

4.5.1 The `ch.dongxi.xpilot.legacy.p2p.message` package

The `ch.dongxi.xpilot.legacy.p2p.message` package contains a class for every type of message exchanged between nodes. `ch.dongxi.xpilot.legacy.p2p.message.Message.Type` is an `enum` that lists all these types.

The `ch.dongxi.xpilot.legacy.p2p.message.MessageMarshaller` is used to convert messages to or form a byte array.

4.6 The `ch.dongxi.xpilot.pulsar.*` classes

The `ch.dongxi.xpilot.pulsar` package provides the classes that act as a layer between *XP2Pilot* and *Pulsar* and the classes that are needed for the configuration of *Pulsar*.

The `Dispatcher` class delivers incoming messages to the appropriate handler (`PieceManager`, `ObjectManager`, ...).

The `GameService` class is the interface between*XP2Pilot* and *Pulsar*. It passes incoming messages to the `Dispatcher` for delivery, and provides methods to send messages, to determine if the node owns a given ID, or to find the first given number of nodes responsible for an ID.

`JoinService` is a wrapper around `org.pulsar.protocol.entrypoint.EntryPointService` making it easier to use it as a *Spring* bean.
`LocalService` does the same for `org.pulsar.protocol.local.LocalService`.

A `PulsarMessage` is a container object for a `ch.dongxi.xpilot.legacy.p2p.message.Message`. It shields *XP2Pilot* from the specific way Pulsar represents a message.

[3]DataTransferObject.

4.7 The `ch.dongxi.xpilot.tasks.*` classes

`ClientLoop` is a `org.pulsar.core.scheduler.IJob` that runs at the `FREQUENCY` defined in `ch.dongxi.xpilot.game.Game`. It handles the commands of the user. (*see 4.10*)

`ServerLoop` is also a `IJob` and runs at the same frequency.
It calls `objectManager.updateObjects();` and `pieceManager.sendRemoteUpdates();`

4.8 The `ch.dongxi.xpilot.view.*` classes

The `ch.dongxi.xpilot.view` contains a GUI that can be used to inspect the internal data structures of a node. (*Fig. 4.1*)

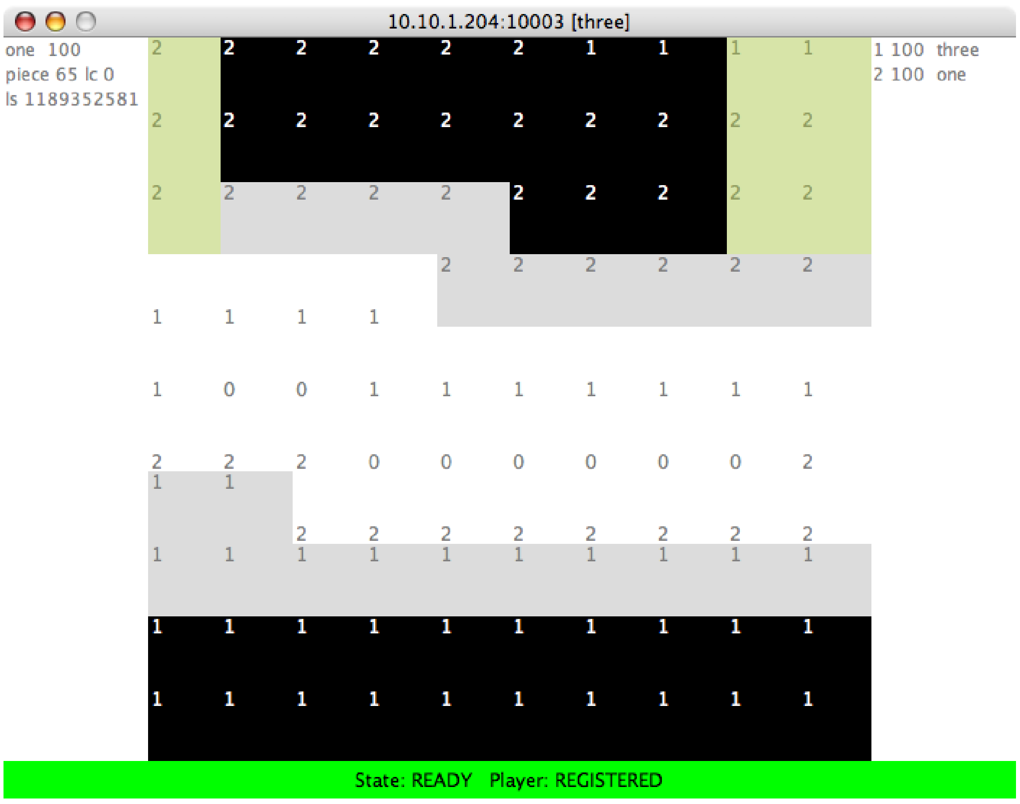


Figure 4.1: Screenshot of the game inspector window. The title bar indicates the node's IP address and port number and the user's nickname. The status bar shows the network state and player state (*see 4.2*). The square in the center represents the world, divided in 10 x 10 pieces (*see 3.4.1*). White pieces are in the *Local* state, grey ones in the *Remote* state, and black ones are *Void* (*see 3.5*). The active zone (*see 3.4.3*) appears in light green. The number in the bottom left corner of the white pieces indicates the number of subscribers. The number in the top left corner of other pieces indicate if the node has a backup for a given piece, and if available, the level of this backup (*see 3.4.1*). The left column displays the content of the registry (*see 3.8*), here with only one entry. The right column displays the score and nickname of known users.

4.9 The `ch.dongxi.xpilot.world.*` classes

The `ch.dongxi.xpilot.world` package provides classes for ships, bullets, walls, positions, and pieces (`WorldPiece`).

The `World` class contains the list of all `WorldPieces`. A piece can be accessed by its number, by its row and column coordinate, or by giving a position it contains. It declares the following constants: `PIECE_COUNT`, `ROW_COUNT`, `COLUMN_COUNT`, `PIECE_HEIGHT`, `PIECE_WIDTH`, `WORLD_HEIGHT` and `WORLD_WIDTH`.

4.10 Threads

XP2Pilot runs with only three threads:

- One thread (`ch.dongxi.xpilot.legacy.net.PacketInputThread`) reads the packets sent by the client, updates the connection information and enqueues the ship commands in a `java.util.concurrent.ConcurrentLinkedQueue`.[4]
- The shutdown hook is invocated by the virtual machine when the user quits the game and thus run only a very short time. (*see 4.2*)
- All other tasks are implemented as `org.pulsar.core.scheduler.IJob` and run inside the thread of the `org.pulsar.core.scheduler.Scheduler`.

[4]which are then dequeued by the `ClientLoop` task (*see 4.7*)

Chapter 5

Conclusion

Although not promised to a bright commercial future, nor having the pretension to be the most fun game ever created, *XP2Pilot* is nevertheless a successful implementation of a peer-to-peer game. It is not only a distributed but also a fully decentralized system. It adapts to the changing overlay network and recovers from inconsistencies with minimal inconvenience for the user.

The transition from one server node to the next one is almost unnoticeable, thus achieving the fluidity required by a realtime game.

The chosen design provides an efficient anti-cheating measure, i.e. the registrar, which hinders the user to locally manipulate its score.

Furthermore, we plan to reuse the solutions and concepts developed for *XP2Pilot* to build a distributed shared clipboard application, which could be used as a model for other peer-to-peer collaboration tools.

Related and Future Work

Many techniques developed on a practical level for this project can be found in solutions described by [10] on a more theoretical level. We divided the world into `pieces` and described an active zone which they called `region`. The essential task of a node as for its server role is managing interactions between objects, which is what their `coordinator` does.

They used *Scribe* [11], an application level multicast infrastructure, to reduce network traffic, whereas *XP2Pilot* could exploit *Pulsar*'s streaming capability. The challenge resides in the tradeoff between reduced bandwidth versus larger latency.

They conclude that building a distributed peer-to-peer game is a feasible option to achieve better scalability. We tend to agree, but like to add that building a fully decentralized and cheat-proof game might prove impossible for a realtime game.[1]

Adding features to *XP2Pilot* to make it more like (or even better than) the original game, might not be the most rewarding undertaking. On the other hand, building on *XP2Pilot*, the implications of cheating on a decentralized peer-to-peer system could be further explored, and new measures to prevent it could be designed [9].

[1] Although a secure implementation of a turn based game like poker should be possible, based on similar cryptographic primitives as in distributed voting protocols.

Finding and adapting a form of week cryptography, secure enough for a peer-to-peer game or any other peer-to-peer system with timing constrains, would be a further challenge.

When considering an environment like the local area network, where implicit trust or security is given, one could explore if *XP2Pilot*'s method of distributing responsibility among nodes based on an underlying DHT could be used for other applications, in order to provide on the application layer what *zeroconf*[2] aims at doing on the network layer.

[2]http://www.zeroconf.org/

List of Figures

Bibliography

[1] Bittorrent protocol specification v1.0, July 2006.

[2] Remo Meier. Peer-to-peer live streaming. Master's thesis, ETH Zurich, 2007.

[3] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM '01 Conference*, 2001.

[4] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, August 2001.

[5] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2001.

[6] Katherine L. Morse. Interest management in large-scale distributed simulations. Technical Report ICS-TR-96-27, Department of Information & Computer Science, University of California, Irvine, 1996.

[7] Jeffrey Pang, Frank Uyeda, and Jacob R. Lorch. Scaling peer-to-peer games in low-bandwidth environments. *International Workshop on Peer-to-Peer Systems (IPTPS)*, 2007.

[8] Markus Bylund and Fredrik Espinoza. Testing and demonstrating context-aware services with quake III arena. *Communications of the ACM*, 2002.

[9] Nathaniel E. Baughman, Marc Liberatore, and Brian Neil Levine. Cheat-proof playout for centralized and peer-to-peer gaming. Technical report, Dept. of Computer Science, University of Massachusetts, 2007.

[10] Björn Knutsson, Honghui Lu, Wei Xu, and Bryan Hopkins. Peer-to-peer support for massively multiplayer games. Technical report, Department of Computer and Information Science, University of Pennsylvania, 2004.

[11] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC)*, 2002.

Appendix A

Beans dependency graph

This graph was generated with the Spring IDE Eclipse plugin.