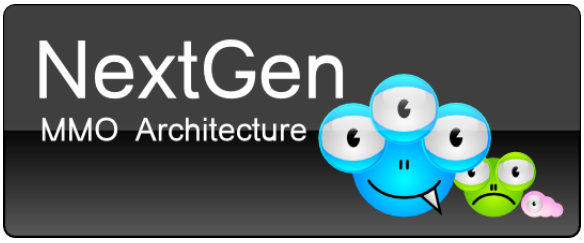
The IT-University of Göteborg

Next Generation MMO Architecture



Author:
Christian Flodihn

Supervisor:
Mattias Klang

January 13, 2010

Abstract

This article presents an implementation of an MMO architecture that takes full advantage of micro threads and multi-core processors to press the server to handle as many players as possible. The solution is to implement a server cluster in Erlang that splits the world into subareas.

I Introduction

MMOG's (Massive Multiple Online Game) are games that let thousands or even millions of people play the same game. A next generation MMO should support an highly dynamic game design, letting the games designers free of the bonds imposed by technical obstacles. Achterbosch, Pierce, and Simmons (2008) investigated what is expected by the next generation of MMO's by asking a group of 122 people . When questioned about what features that was desirable the respondents brought up a number of ideas:

> Rather than players affecting the world, as indicated in Section 6.2.1, many respondents suggested instead that the world change on its own via natural environmental occurrences. The examples include fires that destroy forests, floods that drown the land, the growth of vegetation, and the decay of man-made structures and paths, and so on. This would create a much more immersive world, just as the simple addition of night and day cycles and changes in weather have in the past. The burned forests could eventually regrow, floods could subside, and NPCs (Non Playable Characters) could maintain decaying areas so that the world does not change so dramatically that entire quests, NPCs, and events have to be altered.

Another interesting aspect was suggested but deemed as unfeasible:

> Many respondents believe that their characters should be able to make an impact in the game world that would completely change it forever. In many current MMOs, creatures and NPCs that are part of a quest are reset each time a player completes the quest so that the next person can attempt to do the same. This is commonly referred to as respawning. When a creature that was killed is reset in the world in full health it is said to have respawned. One respondent suggested that the developers permanently alter the world so that creatures or quests did not respawn. The problem is that sooner or later there would be no quests left to complete and no monsters left to kill, as the players would have completed or killed them all. The developer would have to constantly create new content quicker than the players could complete it. This is simply not feasible.

Having game developers constantly creating new content seem unfeasible but with an dynamic game design the world could be automatically searched for opportunities to create quests.

Dynamic NPC's is also a technically challenging feature desired by the respondents:

> Respondents said that NPCs were too static, hence to help with game immersion the NPCs should go about some daily routines, as they do in single-player games like Oblivion and the Gothic series. In most cases, friendly NPCs stand around day and night waiting for players to talk to them. In such cases they have only one purpose, be it a quick one-line statement, a store to buy items, or to give the player a quest. Enemy NPCs tend to stand around waiting to be killed. One respondent suggested that "NPC[s] could be hired to do a range of task[s] from resource gathering to guarding buildings" According to some respondents, the NPC would then have a purpose other than just standing around. Another respondent said that players should have the option to play as NPC enemies for small intervals of time, to make them less predictable, like a dungeon master in Dungeons & Dragons for instance.

The next generation of MMO features requires a state of the art architecture that is allows dynamic game design, distributes and scales. This is a cumbersome task in most programming languages, new tools and programming languages need to be used that better supports developing servers with these requirements. If this could be achieved the players would have a rich experience that makes them feel like they could make impact in a world that is under continuing change.

II Background

There are basically two different types of MMOG architectures, server-client and peer to peer.

A *Client/Server Architectures*

The centralized server-client implementation discussed by (Cai, Xavier, Turner, and Lee, 2002) and (Gil, Tavares, and Roque, 2005) is the most common implemented by todays MMO's. Every player connects with a client to the game servers. The server may verify everything the client sends and may prevent cheating. It is also fairly easy to keep a coherent state of the world, letting the game server handle all events. The problem with this implementation is that when enough players connect, the server will run out of hardware resources and refuse more connections. The most common solution to this problem is called sharding, the game is copied to a number shard servers which houses a couple of thousands players and maintains its own state of the world. This has the negative effect that the player base is divided in isolated groups on the servers, meaning if two friends want to play together they must be on same same server. Another problem is balancing the population, if the server has to many players, people have to wait in a queue to login. If the server has to few players, people might have a hard time finding other people to play with, making the gameplay suffer.

B *Peer to Peer Architectures*

The decentralized peer to peer implementation suggested by (Hampel, Bopp, and Hinn, 2007) and (el Rhalibi and Merabti, 2005) let each client become a server of a geographically limited area. This avoids the scalability problem with centralized servers but highly increases the risk of cheating and increases complexity in terms of maintaining a coherent state for all players.

C *Erlang*

Erlang is a concurrent langauge with built-in distribution, instead of having a main loop that runs continuously Erlang creates a light-weight process for each "object" or task. Each process has its own main loop. An ordinary desktop computer can create hundred of thousands of processes, depending on the hardware. Bundled with Erlang is a platform called OTP (Open Telecom Platform). OTP provides general design patterns for building servers, state machines, fault-tolerance and performing hot code upgrades. Erlang also has its own distributed database called Mnesia. If the architecture would be implemented in Java or C++ all these features would have to be developed from scratch or supplied by third party libraries. Erlang code is organized inside files called modules in which functions are written.

Stated by (Nyström, 2004), just choosing Erlang as implementation language will give a significant performance boost for the servers by using light-weight processess.

D *Theoretical Studies*

There are lot of research for improving MMO's during the recent popularity in these games. The most relevant for this architecture is the research regarding dividing the world into sub areas. This researched is discussed and criticized in this section. There are two commercial projects that using the technique of splitting the world into sub areas, Eve Online (CCP) and Second Life (Linden Lab).

Research papers such as (Chen, Wu, Knutsson, Lu, and Amza, 2005), (Yamamoto, Murat, Yasumoto, and Ito, 2005) and (Bossche, Verdickt, and Vleeschauwer, 2006) proposes dividing the game into sub areas to allow the architecture to scale. When a sub area becomes to loaded it is suggested that it is moved to new hardware. This imposes a limitation in the scalability, one sub area can never host more players than one computer is capable of. By assigning the computers to sub areas instead removes this limitation. What is required is that the

state of the sub area must be distributed across the assigned computers.

E *Commercial Projects*

EvE Online is a game taking place in space. Its architecture divides the world into sub areas (star systems). The cluster contains blade servers, also known as nodes. Each star system resides on a node, several star systems can share a node. The players are not seamlessly transferred between the nodes, when a player "jumps" to another star system a loading screen appears while the transfer takes place. About 60 players engaging in combat has been reported enough to take down a node. If the node is moved to a dedicated server it can handle about 1000 combating or trading players. The developers have revealed that they have problem using all CPU-cores on the machines. About 50% of the cores in the cluster is inactive.«insert ref» This due to the imperative design of Python, the programming language the Eve Servers are implemented in. Python has something the Python developers call "The Global Interpreter Lock" (Lingorm, 2008), it keeps Python from using multiple cores effectively without major internal redesigns.

Second Life has similar architecture of area servers which transfer players seamlessly as they move through the world. There are no servers in front of the cluster responsible for communication with the client. The client keeps four connections, one to each of the four nearest area servers. The area server streams all content in real time to the players, no content is saved on disk which make the client very small.

III Proposed Architecture

As shown in Figure 1, the architecture has four type of servers spread across a cluster of computers, the connection, account, character and area servers. The servers may run on different computers in the same LAN (Local Area Network), this is however not required, all the different servers can be configured to run on the same computer. The server software is implemented in Erlang because of the concurrent, fault-tolerant and distributed nature of the language. The clients connects through the Internet to the connection server. The connection server handles the communation between the client and the other servers. The account server contains the account information that is used to validate logins. On the character server, all the players characters are stored, characters are avatars or entities that represent the players in the game. The area servers should function like a grid of geographically connected areas, illustrated in Figure 2, together they are the world that the players interact with. Each server run in an Erlang node which consists of the node name, the character "@" plus the computer hostname. For example, complete node name is start_area@myserver, where start_area is the node name and myserver is the hostname. The node name is used to comunicate between servers running on different computers.

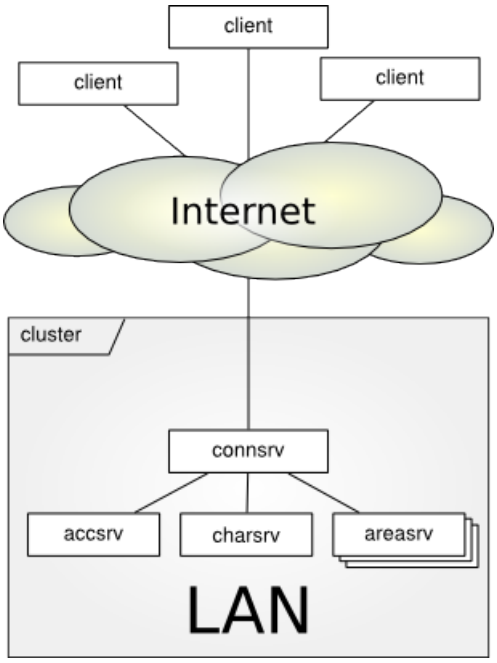


Figure 1: Architecture Overview

A *Client*

The client is the program that runs on the players computer. It makes a connection through the Internet to the connection server. The client is an important part of the system but is not a part of the architecture and could be implemented in any language as long as it can establish a TCP connection to send and receive data from the connec-

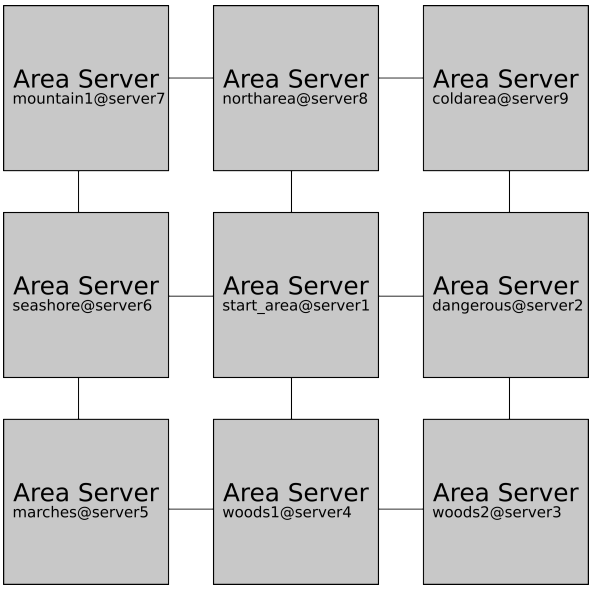


Figure 2: Area Server Grid

tion server. The client's purpose is to represent the world for the player, either through a text-based or grahical interface.

B *Connection Server*

The connection server waits for connections from clients. The connection server serves as the front-end for the cluster. The clients never communicate directly with the other servers, all communications to and from the clients goes through the connection server. For each client that connects a connection process is created. The connections processes are a simple state machines that moves between four states:

- Connected: The client has established a connection but not sent any further commands.
- Lobby: The client has successfully logged in with an account and is browsing or creating entities which to enter the game.
- Playing: The client has logged in with an entity and is currently playing.
- Connection Lost: The client has lost the connection. The connection server keeps the player active, giving him or her a chance to login again and continue to play. When a time-out is reached the connections servers stops the session and terminates.

C *Account Server*

Show in Figure 3, the account server stores all user accounts in a Mnesia database. It provides ability to create and delete accounts. It also have a lookup feature where an account name and password can be validated.

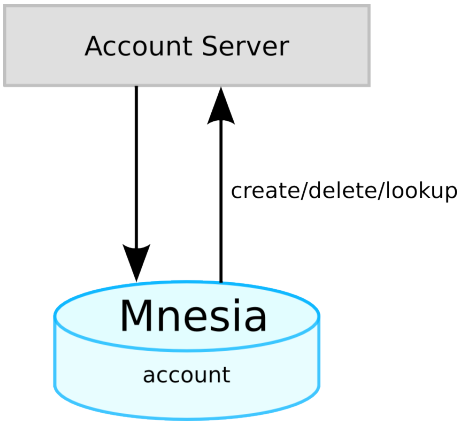


Figure 3: Account Server

D *Character Server*

As illustrated in Figure 4 the character server stores all characters. Every account can create new characters which is used to login to the world. The server provides functions to save and create new characters.

E *Area Server*

The area server is the most complicated server in the cluster. It manages a geographical area of the world. For each object or character that is loaded in the area server, a process is started which keeps its state, this is depicted in Figure 5. Each process has a process id, generated by Erlang and a custom generated id for unique identification in the world. The unique id and process id is stored in a database which is used to search objects.

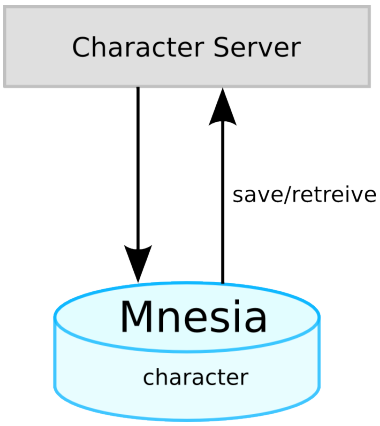


Figure 4: Character Server

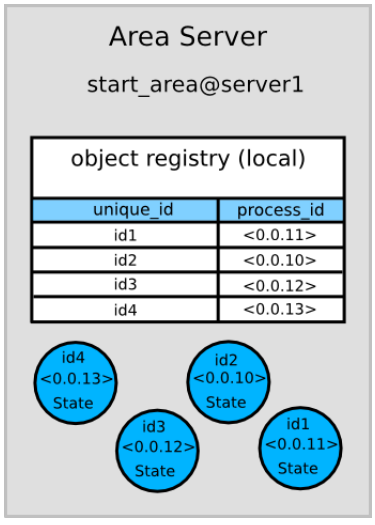


Figure 5: Area Server

F *Dynamic Features*

As shown in 6, the area server has a modular build. Its core only provides a simple event registration and the ability to start and stopp libraries. These libraries in turn provides all the functionality of the area server. The libraries can be compiled, recompiled, started and stopped during runtime. These provides the ability to develope and test new libraries, such as a combat system whithout restarting the server.

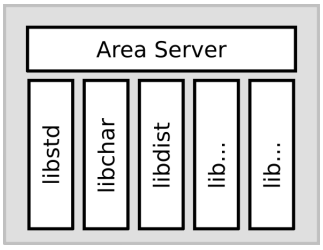


Figure 6: Modular Area Server Build

The architecture comes with three default libraries:

- libstd: This library provides basic functionality such as registering/retrieving objects in the object registry and hot code upgrading.
- libchar: This library allows characters to login on the area server.
- libdist: This library allows an area server to distribute across several computers in the cluster.

Starting a library is done through the library supervisor lib_sup. Note that instead of the starting the libraries directly their supervisors are started, which in turn starts the library and any other process that is needed for the library to function. Thus the module libstd_sup is supplied to the start function instead of libstd. Supervisors in Erlang is process that monitor other processes, if a process crashes the supervisor will restart it. The command to start a library is shown below:

```
lib_sup:start(libstd_sup).
=PROGRESS REPORT==== 7-Jul-2009::12:34:09
supervisor: {local,libstd_sup}
started: [{pid,<0.270.0>},
    {name,'LibStd'},
    {mfa,{libstd,start_link,[std_funs]}},
    {restart_type,permanent},
    {shutdown,2000},
    {child_type,worker}]

=PROGRESS REPORT==== 7-Jul-2009::12:34:09
supervisor: {local,libstd_sup}
started: [{pid,<0.271.0>},
    {name,'ObjectSupervisor'},
    {mfa,{obj_sup,start_link,[]}},
    {restart_type,permanent},
    {shutdown,2000},
    {child_type,supervisor}]

=PROGRESS REPORT==== 7-Jul-2009::12:34:09
supervisor: {local,lib_sup}
started: [{pid,<0.269.0>},
    {name,libstd_sup},
    {mfa,{libstd_sup,start_link,[]}},
    {restart_type,transient},
    {shutdown,infinity},
    {child_type,supervisor}]
{ok,<0.269.0>}
```

A library can also be stopped with the following command:

```
lib_sup:stop(libstd_sup).
ok
```

Another dynamic feature is called hot code swapping. This enables Erlang update code while it is running. A helper function for this, named upgrade has been added to the library libstd. As an example, after making some changes in the module libchar and the new code has sucessfully compiled, upgrading the is done with the following command:

```
libstd:upgrade(libchar).
{module, libchar}
```

Erlang allows two version of the code to run at the same time. When the upgrade function is called with an module as argument, Erlang marks the currently running code as old and the upgraded as current. After the upgrade, the current (new) code will be used. Any process still executing old code will be upgraded next function call.

G *Distribution*

Several computers can manage the same area server. As shown Figure 7, two computers named server1 and server2 runs each own node called start_area. Running the command below, from

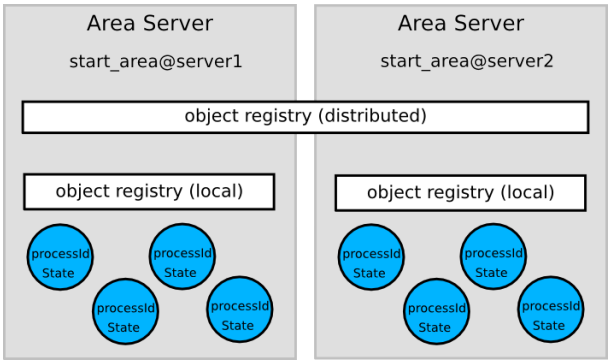


Figure 7: Area Server Distribution

the node start_area@server2 will connect it to the distributed object registry at start_area@server1:

```
libstd:join(start_area@server1).
{ok, [start_area@server1]}
```

The distributed object registry is a Mnesia database that store the same information as the local object registry, with the difference that it contains not just the local processes but all processes in all area servers managing the same area. If start_area@server2 writes data to the distributed object registry, Erlang will propagate the change to start_area@server1 as well. Erlang process id's are location transparent, meaning when they communicate it does not matter if the process runs on the same local machine or another computer in the network. This allow both start_area@server1 and start_area@server2 to operate like the same area server with the distributed object registry that link them together.

G.1 *Load balancing*

The library libdist provides automatic load distribution features. A distributed Mnesia table is up-

dated with the load of all area servers managing the same area. The distribution library checks the on the local area server node against the least lodaded area server node in the table. If the diffrence is equal or greather then 1, the node starts to migrate half its processes to the least loaded area server node.

G.2 *Migration*

Adding distribution made it possible to migrate characters from one server to another during runtime without the client noticing. How the migration of objects takes place is explained below;

Illustrated in Figure 8 an object receives a message that it should migrate to another node. This message is usually sent from the distribution library, libdist but can also be sent manually with the following command from the node:

```
ProcessId ! {migrate, start_area@server1}.
```

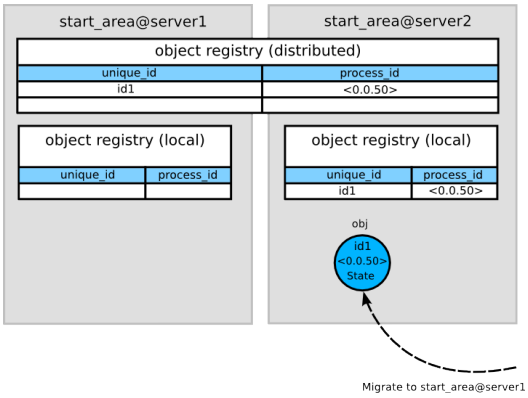


Figure 8: Object Migration Step 1

As shown in figure 9, when the migration messages has been received, the objects current process starts a new process with the same unique id and state on the node it should migrate to. The new process id is returned to the old process on start_area@server2. When the new process is created it writes over the current entry in the global registry with its own process id, thus all new messages should be sent to the itself.

Depicted in Figure 10, at this point if any messages still finds its way to the old process, it is relayed to the new process.

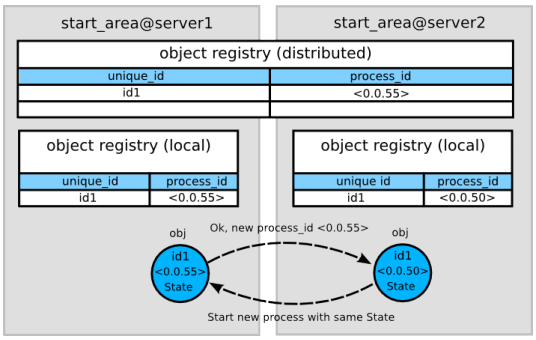


Figure 9: Object Migration Step 2

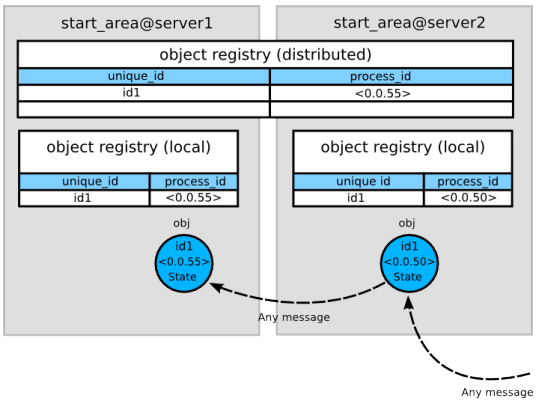


Figure 10: Object Migration Step 3

Shown in Figure 11, if the old process does not receive a new message for one second it kills itself and the migration is considered successful.

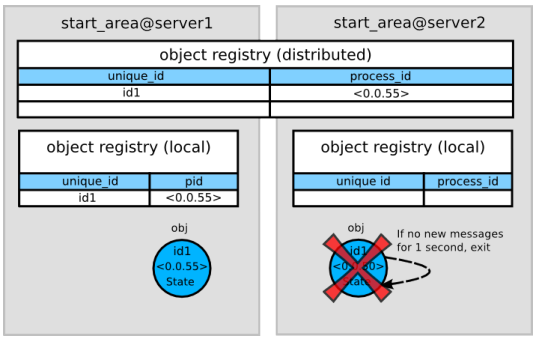


Figure 11: Object Migration Step 4

IV Results

The server used in the tests has an AMD Phenom(tm) II X4 940 Processor and 8 GB RAM. The size of the area is 10 000 meters, the quad tree split the world into 256 quads resulting in a quad size of 625 meters.

In the test, robot objects were created in the area server. The robots move back and forward changing direction about every 7.5 seconds (5 + random 5 seconds). Each movement update is sent to all other objects in the same quad, this means the number of messages in the system increases exponentially compared to the number of robots in the system.

A *CPU Load*

Figure 12 shows the load of all cores when the number of robots increases in the system. The CPU load increases exponentially and at 24 000 concurrent robots all cores are loaded near 100%. With 24 000 robots, about 1178 messages per second is processed in each quad, a total of 301 568 messages per second in the whole area server.

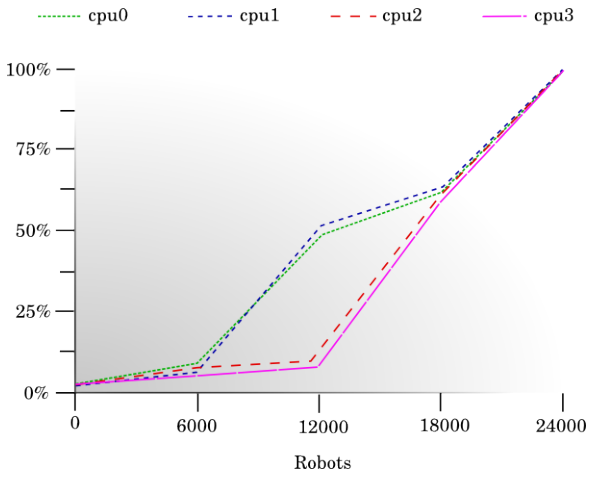


Figure 12: CPU Load

B *Memory Load*

In Figure 13 the memory consumption of the connection, account, character and area server is shown. The area server memory usage increases linear to the amount of robots. With 24 000 robots, the area server used 619 MB of memory while the connection, area and character server remained constant between 40 to 43 MB.

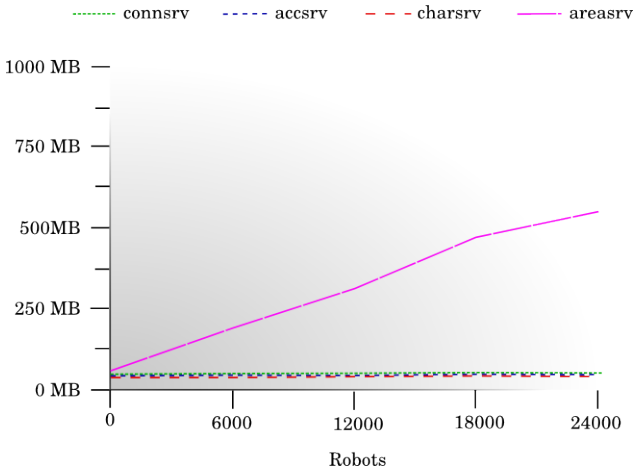


Figure 13: Memory Load

C *Visual Results*

When logging in with a client, about 60-80 robots was counted. Visual timing was performed to check any obvious delay. A few random robots was picked out at different load levels and their movement changes were timed with an IPhone. At 6000 robots in the server, the average movement change was 7.1 seconds (I should measure 12 000 and 24 000 too, but make a log function in the robot instead of manual timing).

V Discussion

Testing thousands of clients in realistic envionment is a problem since it hard to get access to such high number of computers. Running thousands clients from the same computer may create bottlenecks that does not exist when each client have their own computer. The test computers were all connected to the same local area network, this will probably not be the case in a producation environment. The response times (I have not included these in the test yet) will probably increase when connecting from a arbitary Internet connections around the world. In this case the geographical location of the

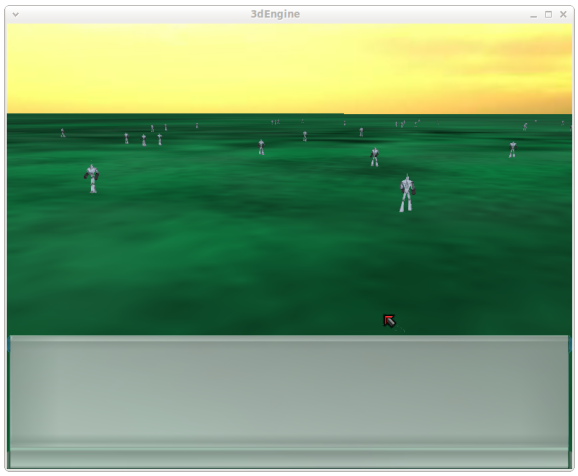


Figure 14: Screenshot

server cluster will also affect the response time of the clients.

The memory consumption of the area server will probably rise when game specific data is added to the processes state. During the testing the processes in the area server had a minimal state sufficent to manage the character login and migration procedures.

The account and character login sequence is not safe since there is no check to detect in an account or character is already logged in, nor or if a character belongs to the account that requested the login.

Dividing the world into area servers might not be the best approach. A better way would be a model the reality, making a hierarky with a universe server, branching out in a galaxy servers which contain sector/solar system servers which in turn contain planet servers that are divided into area servers. This approach would make it easy to distribute messages from a top down approach. This would also easy interstellar travel and even seamless movement from a planet's surface to its surrounding space.

VI Conclusions and Future Work

What needs to be implemented is a system for detecting when objects near the border of an area server should be migrated to the neighbouring server. This will allow creation of a grid that form the shape of the world.

The load balancing system could be even more advanced, at this stage a server has to manually be connected. The ideal would be to have a number of slave servers which is automatically assigned to different areas depending how much load they have.

The architecture in the article is a platform that supports a dynamic game design and with some basic load balancing and the ability to scale horizontally. The architecture is capable of serving many thousands of concurrent players on inexpensive hardware. This concludes that this architecture would be a good platform to further develop next generation of massive multiplayer online games.

References

Leigh Achterbosch, Robyn Pierce, and Gregory Simmons. Massively multiplayer online role-playing games: The past, present, and future. 2008.

Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

Bruno Van Den Bossche, Tom Verdickt, and Bart De Vleeschauwer. A platform for dynamic microcell redeployment in massively multiplayer online games. 2006.

Wentong Cai, Percival Xavier, Stephen J. Turner, and Bu-Sung Lee. A scalable achitecture for supporting interactive games on the internet. 2002.

CCP. Eve online. (http://www.eve-online.com/).

Jin Chen, Margaret Delap Baohua Wu, Björn Knutsson, Honghui Lu, and Cristiana Amza. Locality aware dynamic load management for massively multiplayer games. 2005.

Abdennour el Rhalibi and Madjid Merabti. Agents-based modeling for a peer-to-peer. 2005.

Rui Gil, José Pedro Tavares, and Licinio Roque. Architecting scalability for massively multiplayer online gaming experiences. 2005.

Thorsten Hampel, Thomas Bopp, and Robert Hinn. A peer-to-peer architecture for massive multiplayer online games. 2007.

Linden Lab. Second life. (http://secondlife.com).

CCP Lingorm. Global interpreter lock. (http://myeve.eve-online.com/ingameboard.asp?a=topic&threadID=682229&page=10#285), 2008.

Sven-Olof Nyström. Concurrency in java and in erlang. 2004.

Shinya Yamamoto, Yoshihiro Murat, Keiichi Yasumoto, and Minoru Ito. A distributed event delivery method with load balancing for mmorpg. 2005.

A Migration Code

```
%%% migrate.erl %%%
-module(migrate).

-include("obj.hrl").

-define(MAX_MESSAGES, 1000).

% API
-export([
    init/0,
    migrate/4
    ]).

% handlers
-export([
    ]).

% internal exports
-export([
    migrate_loop/4
    ]).

init() ->
    ok.

migrate(AreaSrv, Type, ObjState, TypeState) ->
    Reply = rpc:call(AreaSrv, obj_sup, start, [Type, TypeState]),
    case Reply of
        {ok, NewPid} ->
            ?MODULE:migrate_loop(AreaSrv, NewPid, 0, ObjState);
        {error, Reason} ->
            {migration_failed, Reason}
    end.

% Migrated objects enter this loop to die gracefully.
migrate_loop(AreaSrv, NewPid, NrMsg, #obj{id=Id} = State) ->
    receive
        Event ->
            NewPid ! Event,
            ?MODULE:migrate_loop(AreaSrv, NewPid, NrMsg + 1, State)
    after 1000 ->
        % If we migrated to a new area, unregister from the shared obj
        % registry.
        case std_funs:area_name() == std_funs:area_name(AreaSrv) of
            true ->
                ok;
            false ->
                dist_funs:unregister_obj(Id)
```

```
        end,
        libstd:unregister_obj(Id),
        %error_logger:info_report([{migration_successful, R}]),
        exit(normal)
    end;

% Force shutdow after 1000 messages
migrate_loop(_AreaSrv, _NewPid, ?MAX_MESSAGES, _State) ->
    error_logger:info_report([{migration_forced,
        {max_messages, ?MAX_MESSAGES}}]),
    exit(normal).
```

B Object Code

```
%%% obj.erl %%%

%%--------------------------------------------------------------------
%% @author Christian Flodihn <christian@flodihn.se>
%% @copyright Christian Flodihn
%% @doc
%% This is the generic object, extended by all objects. It contain
%% functions for get/set properties and area migration.
%%
%% Other objects can call functions the object through the event or
%% async_call function.
%%
%% All functions in an object must return either {reply, Reply, NewState}
%% or {noreply, NewState}, if the object was called throgh the call
%% function the and the function return {reply, Reply, NewState} the tuple
%% {ok, Reply} is sent back to the caller.
%%
%% If the object was called with the call function but returned
%% {noreply, NewState} a tuple {ok, noreply} is sent back to the caller.
%% If the object is called through the async_call function the retuned
%% value is ignored.
%%
%% In all three cases the object continues is execution with the new state.
%%
%% The object can call itself with the function call_self which wraps
%% the values to return {ok, Value, NewState}.
%% @end
%%--------------------------------------------------------------------

-module(obj).
-author("christian@flodihn.se").

%% @headerfile "obj.hrl"
-include("obj.hrl").

%% @headerfile "vec.hrl"
-include("vec.hrl").

-import(obj_loop, [loop_init/2]).
-import(error_logger).
-import(dict).
-import(io).

-import(obj_loop, [has_fun/3]).

% Interface.
-export([
    init/1,
```

```
    post_init/1,
    async_call/2,
    async_call/3,
    async_call/4,
    call/2,
    call/3,
    call_self/2,
    call_self/3
    ]).

% Functions used through inheritance from other modules.
-export([
    event/3,
    event/4,
    set_property/4,
    get_property/3,
    get_id/2,
    is/3,
    transform/3,
    quadtree_assign/2,
    query_entity/2,
    do_anim/3,
    do_anim/4,
    stop_anim/3,
    obj_created/3,
    obj_pos/4,
    obj_dir/4,
    obj_leave/3,
    obj_enter/3,
    obj_anim/5,
    obj_stop_anim/4,
    test/2,
    test/3
    ]).

%%-----------------------------------------------------------------------
%% @spec init(State) -> {ok, NewState}
%% where
%%      State = obj(),
%%      NewState = obj()
%% @doc
%% Initiates the object state.
%% @end
%%-----------------------------------------------------------------------
init(#obj{id=undefined, type=Type} = State) ->
    {ok, Id} = libid.srv:generate_id(),
    libstd.srv:register_obj(Id, self()),
    NewState = State#obj{id=Id},
    {ok, NewState2} = Type:post_init(NewState),
    {ok, NewState2}.
```

```
post_init(State) ->
    {ok, State}.

%%--------------------------------------------------------------------
%% @spec async_call(To, Event) -> ok
%% where
%%      To = pid(),
%%      Event = atom()
%% @doc
%% Same as async_call/4 but with an empty list as arguments and the pid
%% From as self().
%% @end
%%--------------------------------------------------------------------
async_call(To, Event) ->
    async_call(self(), To, Event, []).

%%--------------------------------------------------------------------
%% @spec async_call(From, To, Event) -> ok
%% where
%%      From = pid(),
%%      To = pid(),
%%      Event = atom()
%% @doc
%% When From and To is a pid, this function calls async_call/4 with an
%% empy list as argument.
%% @end
%%--------------------------------------------------------------------
async_call(From, To, Event) when is_pid(From) and is_pid(To) ->
    async_call(From, To, Event, []);

%%--------------------------------------------------------------------
%% @spec async_call(To, Event, Args) -> ok
%% where
%%      To = pid(),
%%      Event = atom()
%%      Args = list()
%% @doc
%% When To is a pid, this will call sync_call/4 with From pid as self().
%% @end
%%--------------------------------------------------------------------
async_call(To, Fun, Args) ->
    async_call(self(), To, Fun, Args).

%%--------------------------------------------------------------------
%% @spec async_call(From, To, Event, Args) -> ok
%% where
%%      To = pid(),
%%      Event = atom(),
%%      Args = list()
```

```
%% @doc
%% Execute an asynchronus call in the object P, calls the function Event
%% with the arguments Args.
%% @end
%%--------------------------------------------------------------------
async_call(From, To, Fun, Args) ->
    To ! {execute, {from, From}, {call, Fun}, {args, Args}},
    ok.


%%--------------------------------------------------------------------
%% @spec call(To, Event) -> {ok, Result} | {error, Reason}
%% where
%%      To = pid(),
%%      Event = atom()
%% @doc
%% Same as sync_call/3 but with an empty list as argument.
%% @end
%%--------------------------------------------------------------------
call(To, Event) ->
    call(To, Event, []).

%%--------------------------------------------------------------------
%% @spec call(To, Event, Args) -> {ok, Result} | {error, Reason}
%% where
%%      To = pid(),
%%      Event = atom(),
%%      Args = list()
%% @doc
%% Executes the function Event with the arguemtns Args in the object To
%% and returns the result.
%% @end
%%--------------------------------------------------------------------
call(To, Fun, Args) ->
    EventId = make_ref(),
    To ! {execute, {from, self()}, {call, Fun}, {args, Args},
        {event_id, EventId}},
    receive
        {EventId, Result} ->
            {ok, Result}
    after 10000 ->
        {error, timeout}
    end.

%%--------------------------------------------------------------------
%% @spec call_self(Fun, State) -> {ok, Reply, NewState} |
%%      {ok, NewState} |{error, Reason}
%% where
%%      Fun = atom(),
%%      Args = list()
```

```
%% @doc
%% Sane as call_self/3 but with an empty list as argument
%% @end
%%--------------------------------------------------------------------
call_self(Fun, State) ->
    call_self(Fun, [], State).

%%--------------------------------------------------------------------
%% @spec call_self(Event, Args, State) -> {ok, Reply, NewState} |
%%      {ok, NewState} |{error, Reason}
%% where
%%      Event = atom(),
%%      Args = list()
%% @doc
%% Used by objects to call to a function in itself.
%% @end
%%--------------------------------------------------------------------
% This function has too many nested cases, fix later.
call_self(Fun, Args, #obj{type=Type} = State) ->
    ArgLen = length(Args) + 2,
    case cache:fetch({Fun, ArgLen}) of
        undefined ->
            case has_fun(Type:module_info(exports), Fun, ArgLen) of
                true ->
                    cache:cache({Fun, ArgLen}, Type),
                    case apply(Type, Fun, [self()] ++ Args ++ [State]) of
                        {reply, Reply, NewState} ->
                            {ok, Reply, NewState};
                        {noreply, NewState} ->
                            {ok, noreply, NewState}
                    end;
                false ->
                    call_self(State#obj.parents, Fun, Args, State)
            end;
        CachedType ->
            case apply(CachedType, Fun, [self()] ++ Args ++ [State]) of
                {reply, Reply, NewState} ->
                    {ok, Reply, NewState};
                {noreply, NewState} ->
                    {ok, noreply, NewState}
            end
    end.

call_self([Parent | Parents], Fun, Args, State) ->
    ArgLen = length(Args) + 2,
    case has_fun(Parent:module_info(exports), Fun, ArgLen) of
        true ->
            cache:cache({Fun, ArgLen}, Parent),
            case apply(Parent, Fun, [self()] ++ Args ++ [State]) of
                {reply, Reply, NewState} ->
```

```
                    {ok, Reply, NewState};
                {noreply, NewState} ->
                    {ok, noreply, NewState}
            end;
        false ->
            call_self(Parents, Fun, Args, State)
   end.

event(From, Fun, State) ->
    event(From, Fun, [], State).

event(_From, Fun, Args, State) ->
    case call_self(get_property, [quad], State) of
        {ok, no_quad, _State} ->
            error_logger:info_report([{obj, event_error, "No quad"}]);
        {ok, Quad, _State} ->
            libtree.srv:event(Quad, Fun, Args)
    end,
    {noreply, State}.

%%--------------------------------------------------------------------
%% @spec set_property(From, Property, Value, State) -> {noreply, State}
%% where
%%      From = pid(),
%%      Property = atom(),
%%      Value = any(),
%%      State = obj()
%% @doc
%% Saves a property Property with the value Value in the object.
%% From is the pid of the process making the call.
%% State is the objects current state.
%% @end
%%--------------------------------------------------------------------
set_property(From, Property, Value, #obj{properties=undefined} = State) ->
    set_property(From, Property, Value, State#obj{properties=dict:new()});

set_property(_From, Property, Value, #obj{properties=Properties} = State) ->
    NewProperties = dict:store(Property, Value, Properties),
    {noreply, State#obj{properties=NewProperties}}.

%%--------------------------------------------------------------------
%% @spec get_property(From, Property, State) -> {reply, Value, State}
%% where
%%      From = pid(),
%%      Property = atom(),
%%      State = obj(),
%%      Value = any()
%
%% @doc
%% Retreives the property Property in the object, returns either the
```

```
%% the property value or if not found, the atom false.
%% @end
%%--------------------------------------------------------------------
get_property(From, Property, #obj{properties=undefined} = State) ->
    get_property(From, Property, State#obj{properties=dict:new()});

get_property(_From, Property, #obj{properties=Properties} = State) ->
    case dict:is_key(Property, Properties) of
        true ->
            Value = dict:fetch(Property, Properties),
            {reply, Value, State};
        false ->
            {reply, undefined, State}
    end.

%%--------------------------------------------------------------------
%% @spec get_id(From, State) -> {reply, Id, State}
%% where
%%      From = pid(),
%%      State = obj()
%% @doc
%% Retrieves the id of the object.
%% @end
%%--------------------------------------------------------------------
get_id(_From, State) ->
    {reply, State#obj.id, State}.

%%--------------------------------------------------------------------
%% @spec is(From, Type, State) -> {reply, true, State} |
%%  {reply, false, State}
%% where
%%      From = pid(),
%%      Type = atom(),
%%      State = obj()
%% @doc
%% Query if the object is a certain type. This is used to determine what
%% type of module an object is extending.
%% For example a player which inherits movable but not god, will return
%% true for the type player and movable but false for god.
%% @end
%%--------------------------------------------------------------------
is(_From, obj, State) ->
    {reply, true, State};

is(_From, _Type, State) ->
    {reply, false, State}.

%%--------------------------------------------------------------------
%% @spec transform(From, NewType, State) -> {noreply, NewState}
%% where
```

```
%%      From = pid(),
%%      Type = atom(),
%%      State = obj()
%% @doc
%% Transforms the object into an object of type Type.
%% @end
%%--------------------------------------------------------------------
transform(_From, NewType, State) ->
    error_logger:info_report([{transforming, State#obj.type, NewType}]),
    {noreply, State#obj{type=NewType}}.

quadtree_assign(_From, #obj{id=Id} = State) ->
    case call_self(get_property, [pos], State) of
        {ok, undefined, _State} ->
            Pid = libtree.srv:assign(Id, self(), #vec{}),
            {ok, _Reply, NewState} = call_self(set_property,
                [quad, Pid], State),
            {noreply, NewState};
        {ok, Pos, _State} ->
            Pid = libtree.srv:assign(Id, self(), Pos),
            {ok, _Reply, NewState} = call_self(set_property,
                [quad, Pid], State),
            {noreply, NewState}
    end.

%%--------------------------------------------------------------------
%% @spec query_entity(From, State) -> {noreply, NewState}
%% where
%%      From = pid(),
%%      State = obj()
%% @doc
%% Queries the object for graphical bound properties, such as mesh,
%% billboard, position and direction. Sends one asyncronus message back
%% the caller for each property.
%% @end
%%--------------------------------------------------------------------
query_entity(From, #obj{id=Id} = State) ->
    %error_logger:info_report([{query_entity, from, From}]),
    case call_self(get_property, [mesh], State) of
        {ok, undefined, _} ->
            pass;
        {ok, Mesh, _} ->
            async_call(From, queried_entity, [{id, Id}, {key, mesh},
                {value, Mesh}])
    end,
    case call_self(get_property, [pos], State) of
        {ok, undefined, _} ->
            pass;
        {ok, Pos, _State} ->
            async_call(From, queried_entity, [{id, Id}, {key, pos},
```

```
                {value, Pos}])
    end,
    case call_self(get_property, [billboard], State) of
        {ok, undefined, _} ->
            pass;
        {ok, Billboard, _} ->
            async_call(From, queried_entity, [{id, Id},
                {key, billboard}, {value, Billboard}])
    end,
    case call_self(get_property, [anim], State) of
        {ok, undefined, _} ->
            pass;
        {ok, Anim, _} ->
            async_call(From, queried_entity, [{id, Id},
                {key, anim}, {value, Anim}])
    end,

    {noreply, State}.

do_anim(From, Anim, State) ->
    do_anim(From, Anim, 0, State).

do_anim(_From, Anim, 0, #obj{id=Id} = State) ->
    {ok, _Reply, NewState} = call_self(set_property,
        [anim, Anim], State),
    call_self(event, [obj_anim, [Id, Anim, 0]], State),
    {noreply, NewState};

do_anim(_From, Anim, Nr, #obj{id=Id} = State) ->
    call_self(event, [obj_anim, [Id, Anim, Nr]], State),
    {noreply, State}.

stop_anim(_From, Anim, #obj{id=Id} = State) ->
    call_self(event, [obj_stop_anim, [Id, Anim]], State),
    {noreply, State}.

%%----------------------------------------------------------------------
%% @spec obj_created(From, Id, State) -> {noreply, NewState}
%% where
%%      From = pid(),
%%      From = id(),
%%      State = obj()
%% @doc
%% The default behaviour is to ignore created objects.
%% @end
%%----------------------------------------------------------------------
obj_created(_From, _Id, State) ->
    %error_logger:info_report([{obj_created, ignored, Id}]),
    {noreply, State}.
```

```
obj_pos(_From, _Id, _Pos, State) ->
    %error_logger:info_report([{obj_dir, ignored, Id, Vec}]),
    {noreply, State}.

%%--------------------------------------------------------------------
%% @spec obj_dir(From, Id, Vec, State) -> {noreply, NewState}
%% where
%%      From = pid(),
%%      From = id(),
%%      State = obj()
%% @doc
%% The default behaviour is to ignore created objects.
%% @end
%%--------------------------------------------------------------------
obj_dir(_From, _Id, _Vec, State) ->
    %error_logger:info_report([{obj_dir, ignored, Id, Vec}]),
    {noreply, State}.

obj_leave(_From, _Id, State) ->
    {noreply, State}.

obj_enter(_From, _Id, State) ->
    {noreply, State}.

obj_anim(_From, _Id, _Nr, _Anim, State) ->
    {noreply, State}.

obj_stop_anim(_From, _Id, _Anim, State) ->
    {noreply, State}.

test(_From, State) ->
    io:format("Test function called successfully.~n"),
    {noreply, State}.

test(_From, Arg, State) ->
    io:format("Test function called successfully with arg: ~p.~n",
        [Arg]),
    {noreply, State}.
```