Joel Partanen

Implementing a Proxy Server for a Massively Multiplayer Online Role-playing Game

Tradenomi
Tietojenkäsittely
Syksy 2018

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä(t): Partanen Joel

Työn nimi: Välityspalvelimen toteuttaminen massiiviseen verkkoroolipeliin

Tutkintonimike: Tradenomi (AMK), tietojenkäsittely

Asiasanat: MMORPG, verkko-ohjelmointi, välityspalvelin

Tässä opinnäytetyössä toteutettiin välityspalvelin kehityksessä olevaa MUTA-peliä varten. Tavoitteena oli, että välityspalvelin kykenisi piilottamaan pelipalvelimen todellisen IP-osoitteen, suodattamaan liikennettä, sekä turvallisesti ohjaamaan viestejä pelaajien ja pelipalvelimen välillä tuottamatta huomattavaa lisäviivettä.

Pääosa opinnäytetyöstä keskittyi välityspalvelimen toteutukseen käytännössä. Ennen toteutusosiota annettiin kuvaus MMORPG-peligenrestä, jonka jälkeen käytiin läpi tiettyjä genrelle teknisesti tärkeitä konseptejä, kuten verkkoliikenteen salaaminen ja skaalautuvat verkkoyhteyksien hallintarajapinnat. Muutoin teoriaan viitatessa referoitiin yleensä erinäisten pelinkehittäjien antamiin lausuntoihin tässä työssä toteutetun ohjelmiston kaltaisista järjestelmistä.

Päätelmässä tultiin tulokseen, että toteutettu ohjelma kykenee piilottamaan pelipalvelimen IP-osoitteen ja suodattamaan verkkoliikennettä pienillä pelaajamäärillä tuottamatta huomattavaa lisäviivettä yksinkertaisessa testiympäristössä, mutta että lisää testausta realistisemmassa ympäristössä vaadittaisiin vastaisuudessa.

KAMK • University of Applied Sciences

Abstract

Author(s): Partanen Joel

Title of the Publication: Implementing a Proxy Server for a Massively Multiplayer Online Role-playing Game

Degree Title: Bachelor of Business, Information Technology

Keywords: MMORPG, network programming, proxy server

In this Bachelor's thesis, a proxy server was implemented for the in-development massively multiplayer online role-playing game (MMORPG) MUTA. The objective was that the proxy server would be capable of hiding the game server's true IP address, filtering traffic, and safely routing messages between players and the game server without introducing noticeable extra latency.

Most of the thesis focused on the practical implementation of the proxy server. Before the implementation part, a description of the MMORPG video game genre was given, and important technical concepts related to the genre were introduced, including things such as encryption and scalable input and output application user interfaces. Otherwise, when theory was referenced, it was mostly anecdotes from various game developers who have publicly talked about the implementations of similar systems.

It was concluded that the implemented program allows for hiding game server IP addresses and filtering network traffic with small amounts of players without introducing perceptible extra latency in a simple testing environment, but that more thorough testing in a more realistic environment was required.

Foreword

Massively multiplayer online role-playing games (MMORPGs) are a technically demanding genre of video games. Often their server-side software consists of multiple complex programs, and unlike single-player games, they must not only make the game look good, but also make it secure and capable of communicating with a server potentially thousands of kilometers away in a way that feels free of latency.

A proxy server sometimes constitutes one of the parts of an MMORPG's server architecture. To protect themselves from denial-of-service (DOS) attacks, online games can hide their game servers' IP addresses by using proxy servers to redirect and filter the traffic between the game server and the players. In this thesis, a proxy server will be implemented with this security perspective in mind.

Sisällys

1 Introduction 1

2 On Massively Multiplayer Online Role-playing Games 2

2.1 Proxy Servers 2

2.1.1 Using Proxy Servers to Mitigate Denial-of-Service Attacks 2

2.1.2 Using Proxy Servers as Load Balancers 3

2.2 MMORPG Architecture 3

2.2.1 Parts of an MMORPG Server 3

2.2.2 Example: Mortal Online 4

2.2.3 Example: Tibia 4

2.3 MUTA 5

2.4 Encryption in MMORPGs 8

2.4.1 Secret-key Cryptography 9

2.4.2 Libsodium 9

2.4.3 Encryption in MUTA 9

2.5 Scalable I/O Interfaces: epoll and Windows IOCP 9

3 Practical Implementation 11

3.1 Research and Planning 11

3.2 Requirements of the Proxy Server 11

3.3 Language and Libraries 11

3.3.1 The netpoll API 12

3.4 Architecture 15

3.4.1 Event-based Architecture 16

3.4.2 The Event Buffer 17

3.4.3 An Example of Posting and Handling an Event 20

3.4.4 Posting Variable Size Events 22

3.5 Modules 23

3.5.1 The Common Module 23

3.5.2 The Client Module 23

3.5.3 The Shard Module 26

3.5.4 Interactions between the Modules 27

3.6 Converting the Shard Server to Work with Proxies 28

3.6.1 The Proxy Module of the Shard Server 28

3.7 Testing 30

3.8 Proxy Server Behavior 30
3.8.1 Forming a Connections with Shard Servers 31
3.8.2 The Proxy-Shard Protocol 31
3.8.3 Forming Connections with Player Clients 32
3.8.4 Routing Packets between Clients and Shards 34
3.8.5 Further Development 35
4 Conclusions 37
Sources 38

List of Symbols

MMORPG

Short for massively multiplayer online role-playing game. A video game genre that features role-playing within a persistent world that hundreds or thousands of players can access simultaneously over the internet.

Proxy server

An internet server whose purpose is to route traffic between two different ends, resulting in an indirect connection between the two sides.

POSIX

The Portable Operating System Interface, POSIX, is a set of standards created for the purpose of maintaining compatibility between different operating systems (Josey, 2017).

Shard

A term used to refer to a single copy of an MMORPG world that usually runs on its own server (Koster, 2009).

1 Introduction

Massively multiplayer online role-playing games are demanding to implement. One of the reasons for the difficulty involved is simply the scale of the products: a single world may simultaneously be home for up to tens of thousands of players (CCP Games, 2016).

A further complicating feature is the fact these games are so closely tied to the internet. Most game developers will probably not care if a player cheats in a singleplayer game, but when the cheating goes online, it will affect other paying customers as well, which can be a problem.

A game that functions on the internet must assume that no data sent by a client may be blindly trusted. If it is trusted, a malicious client may, for example, send incorrect data to crash a server, or slow a server down by sending a constant stream of unnecessary messages that take a lot of time to process. These are known as denial-of-service attacks.

For example, the developers of the game *Tibia* have described a problem where players who wish to hurt another player will attack a game server the target currently plays on with a distributed denial-of-service attack, preventing them from playing (Payer, 2012).

The effects of denial-of-service attacks can be mitigated by using a correctly built proxy server in between the clients and the game server (Dunn, 2018).

In this thesis, a real proxy server will be implemented to act as an intermediary between servers and clients of the MMORPG MUTA. The proxy must also be able to do primitive packet filtering. It must be able to connect to multiple game servers at once, and game servers must be able to simultaneously connect to multiple proxies.

2 On Massively Multiplayer Online Role-playing Games

Massively multiplayer online role-playing games, or MMORPGs for short, are games that incorporate role-playing mechanics in a context where hundreds or thousands of players share a virtual, persistent space. The genre is seen to have developed from MUDs, multi-user dungeons, which came into existence in the 1970s. MUD1, the first multi-user dungeon game, was written in 1978 by Roy Trubshaw and Richard Bartle (Bartle, 2003).

While most MUDs were text-based, MMORPGs, which came into existence in the 1990s, incorporate graphical worlds, often at a massive scale. Meridian 59, released in 1995, is often attributed the title of the first graphical MMORPG: it used sprite-based 3D graphics (Schubert, 2012).

2.1 Proxy Servers

Proxy servers are programs designed to route traffic from one end to another, the result being an indirect connection between two computers. In this subchapter we will be talking about proxy servers from the perspective of MMOPRG development.

2.1.1 Using Proxy Servers to Mitigate Denial-of-Service Attacks

Routing traffic through a proxy allows for hiding the IP address of a service. This can be helpful to prevent direct denial-of-service attacks against an MMORPG game server (Rudy, 2011). Although the attacker can attack the proxies instead, it is possible to deploy multiple proxy servers, increasing the surface the attacker has to cover to bring the whole service down (Dunn, 2018). Additionally, proxy servers can do package filtering to drop invalid packets sent by malicious users, reducing the amount of data the game server has to process.

2.1.2 Using Proxy Servers as Load Balancers

Since proxy servers take away some of the work of client connection management from the service behind them, they might be able to improve the service's performance, assuming connection management was taking a large portion of the service's time previously.

2.2 MMORPG Architecture

Due to their massive nature, and the fact they function exclusively online, MMORPGs have many technical demands other video game genres do not face. Their development cycles are often slow and demanding.

Operating online increases a video game's technical complexity; in addition to a client program, software for the game servers is required, and the client program must now not only do what a normal video game does, such as handling graphics and input, but also handle network connections with the server and incorporate logic to make the gameplay feel smooth despite the inevitable latency between the server and the client. Since players might cheat, the game servers cannot blindly trust the clients either, so cheat prevention becomes necessary.

The business model MMORPGs use is also more technically demanding than that of a single-player video game. Whereas a single-player video game might be sold at a given box price, MMORPGs operate as a service. The game servers must be running for the players to be able to use the product. Often MMORPGs incorporate a pay-by-time subscription model (Schubert, 2012), which also means the service provider must have both, software and work force to deal with players' online purchases of subscriptions.

2.2.1 Parts of an MMORPG Server

As far as MMORPG server architectures go, there isn't a single one that would suit all games, because each product has its own demands. At the simplest, an MMORPG might constitute of two programs: a client and a server. But more often than not this is not the case.

Some parts an MMORPG's server side might include are the following.

- Login server that handles player’s login sessions,
- Game servers, sometimes called shards, that do the game simulation,
- Database servers that are dedicated to caching, saving and loading the persistent data of an MMORPG world,
- Billing and web servers,
- Proxy servers that reroute traffic.

It depends on the game what kind of parts it requires.

2.2.2 Example: Mortal Online

The commercial MMORPG Mortal Online reportedly uses at least the following programs on its server side, in addition to managerial programs (FarmerJ03, 2014).

- ClusterNodes that run gameplay logic such as AI, the world is split up into multiple of these,
- ClusterServer, a server that forwards packets to multiple ClusterNodes and knows all of the game objects in them,
- FrontEnd, the server to which clients actually connect to, and which forwards data to the ClusterServer as well as back to the clients.

In Mortal Online the users do not directly connect to the game server that does the simulation. In fact, there are multiple programs in between, as seen above.

2.2.3 Example: Tibia

Tibia, a 2D MMORPG released in 1997 (MPOGD, 2007), also uses its own set of programs on its server side (Rudy, 2011).

- Login servers,
- Game Servers,
- Dynamic Web Servers,
- Static Web Servers,
- Database Query Managers,

- Payment Server.

In 2011, Matthias Rudy, a developer of the game, stated their new server's architecture would likely include proxy servers between the game servers and the clients. He used the term *dis-patcher* for the proxies (Rudy, 2011).

2.3 MUTA

The practical part of this thesis will be completed as part of the architecture for MUTA, a MMORPG currently in development.

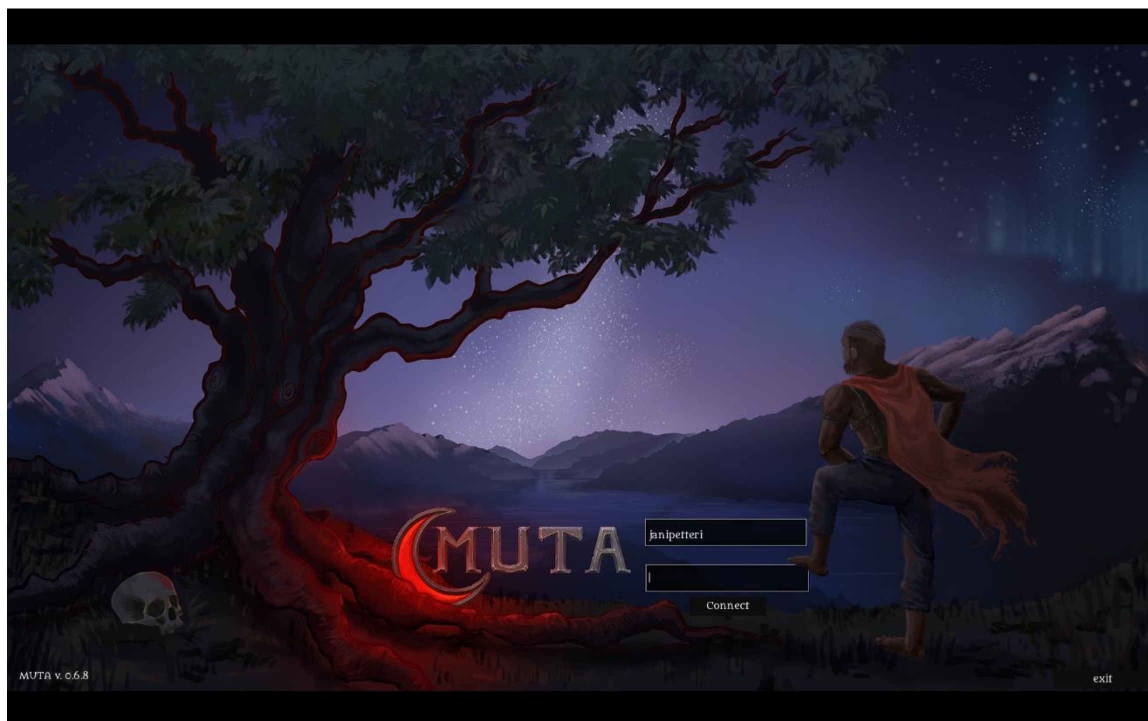


Image 1: MUTA Login screen

MUTA, short for Multi-user Time-wasting Activity, is a MMORPG project started in 2017 by the author of this thesis in collaboration with other KAMK students. The server and the client are free software licensed under the Affero General Public License.

Graphically the game is 2D, presented from an isometric perspective. It's tile-based, meaning objects move on a fixed-size grid, which in this case is 3-dimensional. The game features click-to-move movement, making for a fairly slow-paced experience that doesn't have high demands for low latency. All networking in MUTA is done with TCP.

Image 2: Gameplay early in MUTA’s development

The Initial Server Architecture

At the time of beginning the work on the proxy server written during this thesis, MUTA’s server architecture consisted of three programs.

- Simulation servers: a world is divided into multiple simulation servers that do things such as manage timers and AI,
- Shard server: a central point of connectivity that connects the simulation servers together. Aware of all game objects in the game world. Players directly connect to the shard server. Also handles player logins.
- Database server: a server that handles database queries. The shard server connects to it.

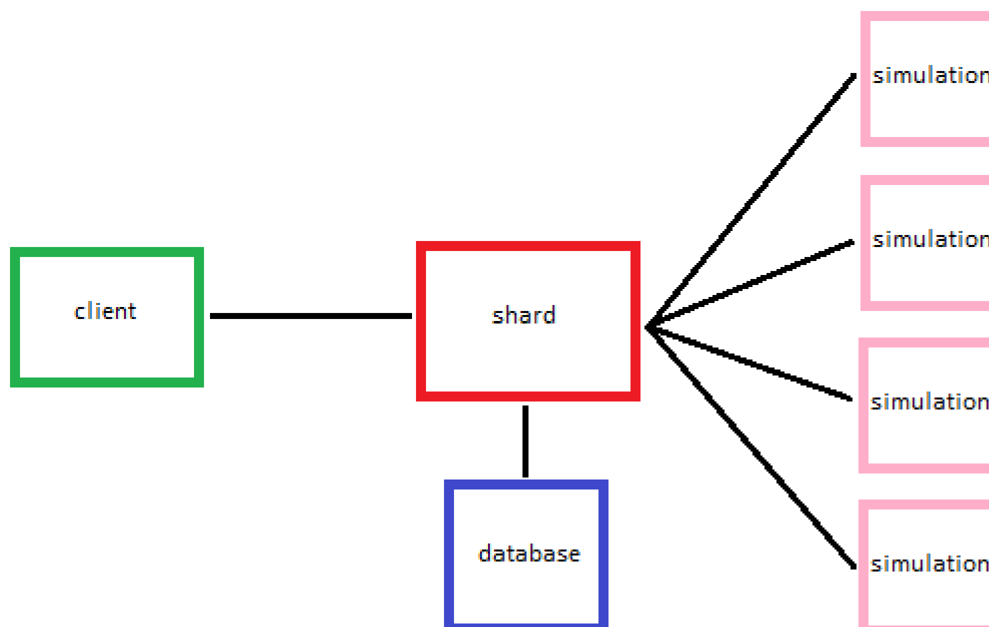


Image 3: MUTA’s initial server architecture

The New Server Architecture

It had been in the plans of the developers of MUTA for a while to rework the ad-hoc architecture that had existed for the game since the beginning of development. It had been decided that a separate login server was needed, as well as proxy servers to provide more security and scalability.

In the new architecture, clients would no longer connect directly to game servers, but would first connect to a login server, from which they would get a ticket to connect to a proxy server. The proxy server would be used to move network traffic between clients and game servers.

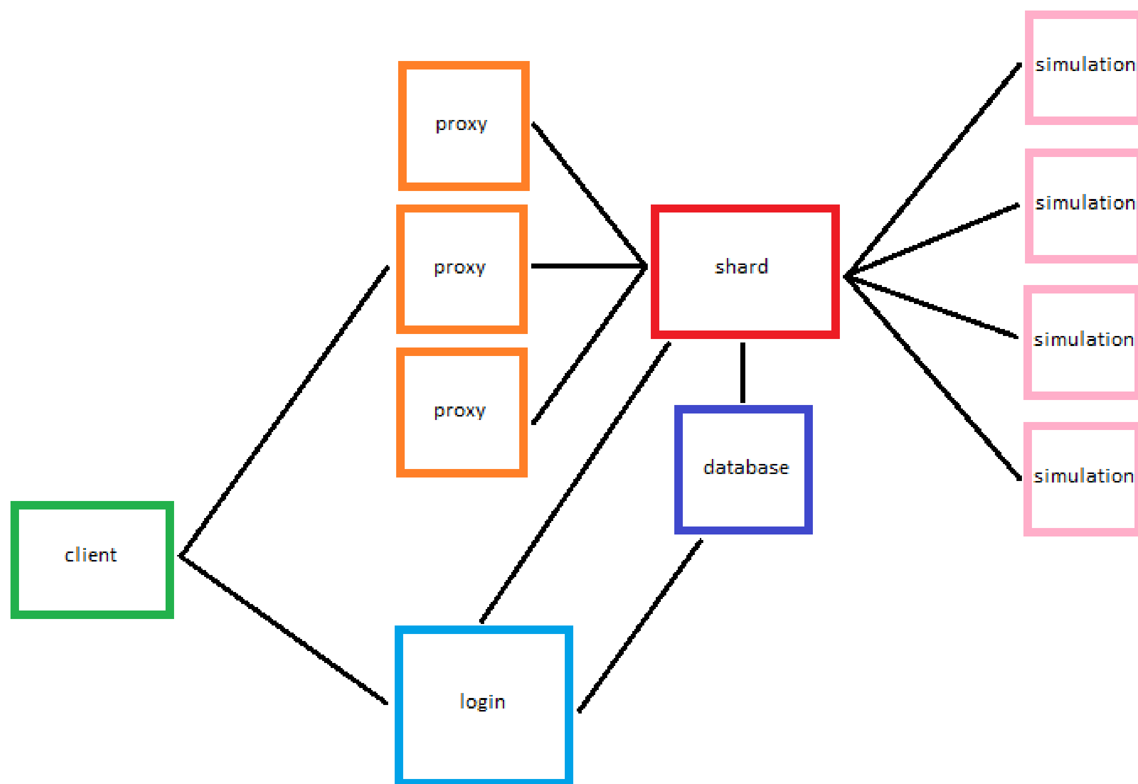


Image 4: MUTA's new server architecture

2.4 Encryption in MMORPGs

Encryption is a procedure in which messages are encoded in such a way only authorized parties are able to read them. In the context of MMORPGs, encryption usually refers to encryption of message streams between a client and a server, or between multiple servers.

Different encryption methods are employed by different games. Some of them are easily broken by hackers (Springer, 2015), others are more secure.

Encryption is important for the purposes of protecting data such as passwords and players' private messages, as it makes less likely a third party will be able to read the data in a man-in-the-middle style attack. It can also be used to detect an attack where another person attempts to impersonate a different client, since the attacker must also know the encryption keys and possibly also the correct spot in the stream of sent data.

2.4.1 Secret-key Cryptography

Secret-key cryptography is a method of encryption where a single key is used to both, encrypt and decrypt data, meaning both ends must know the same key to be able to send or receive, while the key remains secret to outsiders (Denis, 2017).

Because keys must be exchanged before a connection becomes encrypted, methods have been developed to securely exchange keys between two parties. One well-known method for doing so is the Diffie-Hellman key exchange (Rescorla, 1999).

2.4.2 Libsodium

Libsodium is a library for encryption, decryption, signatures, password hashing and other things (Denis, 2017). The library is written in the C language and provides methods for secure key exchange as well as stream encryption.

2.4.3 Encryption in MUTA

MUTA uses Libsodium to implement encryption of network messages. More specifically, the secure key exchange methods based on Diffie-Hellman (Denis, 2017) and the stream encryption API are used.

MUTA implements its own abstractions over Libsodium so that the library’s functions are not directly called in most of the codebase.

2.5 Scalable I/O Interfaces: epoll and Windows IOCP

MMORPG servers must be able to handle thousands of simultaneous connections from player clients. Operating systems have application programming interfaces designed for the efficient management of large amounts of network traffic, but they differ quite a lot from platform to platform.

Linux has the epoll interface. Windows has the I/O completion ports interface. These interfaces differ in their philosophy and are hence difficult to abstract in such a way the same code would be nearly as efficient on both operating systems (George, 2014). Both interfaces are used to build scalable MMORPG servers (Wyatt, 2012).

3 Practical Implementation

3.1 Research and Planning

The proxy server was implemented into the game MUTA, which had been worked on for a little over a year at the time of starting the proxy server project. Initially, MUTA's networking model was simple: clients connected directly to the game server, and the game server also acted as the login server. However, it had been known well beforehand that this simple model would change later, so the program had been somewhat designed with the upcoming changes in mind.

There was one technical talk that greatly inspired starting the work on the proxy server in the first place: Matthias Rudy's talk called "Inside Tibia – The Technical Infrastructure of an MMORPG" from Game Developers' Conference 2011. In this talk, Rudy mentioned he wanted to implement proxy servers in a future game. It was decided MUTA would implement a fairly similar approach.

The concept of a proxy server itself was deemed fairly simple, so little research was required. While at the beginning the game server acted as the login server, a new, separate login server application was also developed during this project, but it is not discussed in this thesis.

3.2 Requirements of the Proxy Server

The requirements laid out for the proxy server were the following.

- It must be possible to have more than a single proxy connected to a shard server,
- It must never be necessary to reveal the IP address of the shard server to clients,
- The proxy server should introduce little more latency in comparison to direct connections between server and client.

3.3 Language and Libraries

Most of the MUTA suite of applications is written in the C language, with the exception of some scripts that are written in Lua. Thus, the proxy server was also to be written in C.

The MUTA codebase uses few external libraries but implements its own common libraries across the different applications it includes. The programs previously written for the server used *netqueue*, MUTA's own abstraction over Windows' I/O Completion Ports and Linux's epoll interface, for dealing with large amounts of network connections. For the proxy, it was decided that good performance on Windows was not important because it was not intended to run on the platform except during development, and thus a new interface, better suited for Linux development was created. The new interface became known as the *netpoll* interface, and while on Linux it uses the efficient epoll API, on Windows it uses the fairly inefficient WSAPoll call. This allows the proxy to run on both, Windows and Linux while primarily focusing on Linux performance-wise.

3.3.1 The netpoll API

The netpoll API was the first thing implemented for the proxy server. It's an extremely thin wrapper over Linux' epoll interface, with some restrictions. It does not support edge-triggered mode on Windows, which on Linux allows notifications to only be generated when a completely new event on a socket happens. The API only supports network sockets in contrast to any file descriptors, and the blocking wait-call cannot be called from multiple threads on the same polling context simultaneously.

The reasons for implementing the API were two-fold: performance and ease-of-use. MUTA's previous *netqueue* API, written for the same purposes of managing multiple network connections at once, had been designed around the Windows model of multiplexed socket handling. In the Windows I/O Completion Ports model, each socket requires a separate buffer in the application's private address space, whereas on Linux, using the epoll API, this is not required since the application itself gets to determine when data is transferred from the operating system into its memory space (George, 2014).

The design also forced some compromises to be made in the use of epoll. For example, on Windows it was not possible to delete sockets from a netqueue-instance before they would be closed. The EPOLLONESHOT flag was also used, due to the differences in message delivery between the Windows and Linux APIs, making the Linux implementation less efficient than it could have been as sockets had to be added back to the netqueue context constantly.

In terms of ease-of-use, the netqueue API was complex to use as it required the provision of multiple callback functions. It also ran on its own internal threads, making it more difficult to reason

about. Hence it was decided that something akin to the poll/epoll APIs was desired, as they leave thread management to the application and are easy to use as well as straight-forward to reason about.

Below is described the public netpoll API, without the operating system specific things.

```
enum netpoll_flag {
    // These bitmasks differ depending on OS
    NETPOLL_READ
    NETPOLL_WRITE
    NETPOLL_ET
    NETPOLL_ERR
    NETPOLL_HUP
};

typedef struct netpoll_t netpoll_t;

union netpoll_data_t {
    void *ptr;
    int fd;
    uint32_t u32;
    uint64_t u64;
};

struct netpoll_event_t {
    uint32_t events;
    union netpoll_data_t data;
};

int netpoll_init(netpoll_t *np);
void netpoll_destroy(netpoll_t *np);
int netpoll_wait(netpoll_t *np, netpoll_event_t *ret_events,
    int max_events, int timeout_ms);
int netpoll_add(netpoll_t *np, socket_t s, netpoll_event_t *e);
int netpoll_del(netpoll_t *np, socket_t s);
```

So, a netpoll instance is initialized with netpoll_init() and destroyed with netpoll_destroy().

Sockets are added to the netpoll to be monitored using netpoll_add(), and deleted with netpoll-del() (the file descriptor type, socket_t, is a typedef for an int on Linux, and SOCKET on Windows). Along with a call to netpoll_add() a pointer to a netpoll_event_t is passed: this structure allows describing flags for what events are to be monitored (reading, writing, etc.) as well as extra flags like NETPOLL_ET which enables edge-triggered behavior on Linux. It also has a union member named “data”, where a pointer or an integer can be stored to uniquely identify the socket once an event is dispatched for it.

netpoll_wait() sleeps until an event is dispatched regarding one of the monitored file descriptors by the operating system, or the time given in the timeout parameter in milliseconds elapses (a

timeout value of -1 means sleeping indefinitely). An event in the case of the proxy server usually means that either new data arrived over the network, or a new connection is coming in. The function will return up to max_events netpoll_event_t structures to the array pointed to by parameter ret_events, and the number of the written events as the function's integer return value. The netpoll_event_t is exactly similar to that of Linux's epoll's struct epoll_event: in fact, on Linux, it's just a typedef.

On Windows, netpoll_wait() uses WSAPoll(), which is analogous to the basic poll() function found in POSIX-compliant operating-systems, and was introduced in Windows Vista (Microsoft, 2018). It is intended to make the process of porting poll()-using programs to Windows easier (Carlin, 2006).

Unlike epoll, WSAPoll() requires constant managing of the monitored file descriptors in application memory, as they must be passed in as an array every time the function is called (parameter fdArray in the function prototype below).

```
int WSAAPI WSAPoll(
  LPWSAPOLLFD fdArray,
  ULONG       fds,
  INT         timeout
);
```

Because of this requirement, sockets registered with the netpoll instance are internally stored in a contiguous array. A separate array, equal in length to the WSAPOLLFD array required by WSAPoll(), is used to store the netpoll_event_t structures. Registration and deletion of sockets with a netpoll are protected by a mutex.

Since WSAPoll() is a blocking call, it must be woken when a new socket is registered with it so that events for the socket will be properly tracked. For this reason, netpoll_add() first adds the new sockets into a queue on the netpoll, keeping the aforementioned mutex locked during the operation. Then it wakes up the WSAPoll instance by sending a TCP message to a dummy socket. A TCP socket is used because during quick testing, sending data to a UDP socket appeared not to wake up WSAPoll().

Because TCP is used for the dummy sockets used to wake up WSAPoll(), a listening socket must be bound during initialization. Because multiple proxy servers may be running on the same machine, each one of them must bind a socket to the dummy socket's port during initialization, but

the listening socket can be closed right after forming the connection. Thus, a simple for-loop is used during initialization: it attempts to bind to the dummy socket's port until success.

Basic usage of the netpoll API looks as follows.

```
netpoll_t   netpoll;
socket_t    read_socket;

...

// Register a socket with the netpoll
netpoll_event_t event;
event.events = NETPOLL_EVENT_READ;
event.data.ptr = &read_socket;
netpoll_add(&netpoll, read_socket, &event);

...

for (;;) {
    netpoll_event_t events[16];
    int num_events = netpoll_wait(&netpoll, events, 16, -1);
    for (int i = 0; i < num_events; ++i) {
        socket_t *socket = events[i].data.ptr;
        ... handle event ...
    }
}
```

3.4 Architecture

Initially the proxy server was designed around a fixed-rate loop, much like video games and other interactive software. After a week of development, it was decided however that this approach had some problems. First, a fixed-rate loop introduces built-in latency, as a network message's handling may be delayed by the fixed length of a frame. Second, this kind of a polling-based approach is wasteful of CPU resources, because if there are no network events to process, the CPU will just be doing busy-work in the fixed-rate loop.

Instead it was decided the server would be based on an *event-based architecture*, or so the architecture came to be called during the project. This was an internal working term, and it was never clarified if the term is really a widely used one. Next, the architecture will be described in detail.

3.4.1 Event-based Architecture

This architecture was come up with to simplify the control flow of the server. The main idea is that most stateful logic, that is, logic that manipulates application state, must be performed on the main thread of the application. Other threads will, instead of handling their own events, push the events into a queue to be processed on the main thread.

Instead of polling for new events at a fast rate, the main thread sleeps until it is given events to handle by other threads. There are two other threads in the application initially: a networking thread that handles incoming data from shard servers, and a networking thread that handles incoming data and connections from players.

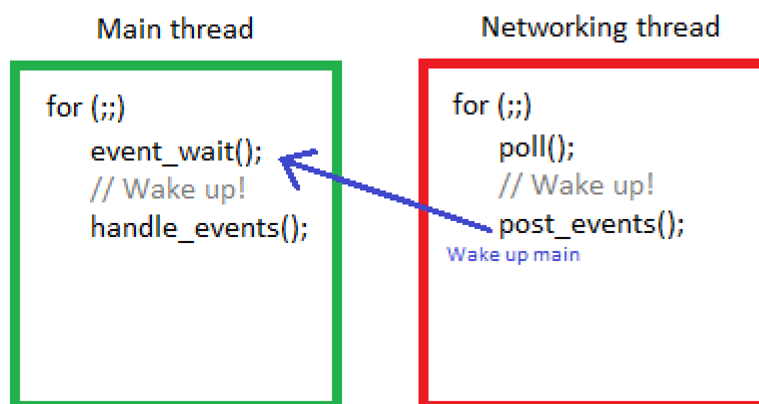


Image 5: A graph showing how the event-based architecture of the proxy server works.

3.4.2 The Event Buffer

The proxy server has three different event types:

- EVENT_READ_SHARD: when a message from a shard server is received,
- EVENT_ACCEPT_CLIENT: when a potential new player client connection has been detected,
- EVENT_READ_CLIENT: when a message from a player client is received.

The data concerning any of the events is stored in a C union. The type member of each structs is an integer enumeration of one of the events defined above.

```
struct read_shard_event_t {
    int      type;
    uint32   shard;
    int      num_bytes;
    uint8    *memory;
};

struct accept_client_event_t {
    int         type;
    socket_t    socket;
    addr_t      address;
};

struct read_client_event_t {
    int      type;
    uint32   client_index;
    uint32   client_id;
    int      num_bytes;
    uint8    *memory;
};
```

```
struct event_t {
    union {
        int                     type;
        read_shard_event_t      read_shard;
        accept_client_event_t   accept_client;
        read_client_event_t     read_client;
    };
};
```

The data structure that events get queued up to is struct event_buf_t, known as an event buffer. The API to using an event buffer looks as follows.

```
typedef struct event_buf_t event_buf_t;

...

void event_init(event_buf_t *buf, uint32 item_size, int32 max);
void event_destroy(event_buf_t *buf);
void event_push(event_buf_t *buf, void *evs, int32 num);
int event_wait(event_buf_t *buf, void *evs, int32 max_evs,
    int timeout_ms);
/* -1 as timeout means infinite */
```

The event_init function takes an item_size parameter. This is the size of the event structure, in this case sizeof(event_t). The reason the size must be passed is that the event buffer API is used in more programs than just the proxy, and sometimes a single program even has multiple buffers with differently sized events in each one of them.

The most important parts of the API are the event_push() and event_wait() functions.

event_wait() takes in a pointer to an array of event structures, the size of the array and a timeout in millliseconds. It sleeps until either the timeout elapses, or new events are pushed to the event buffer. When the function wakes, it writes up to max_evs event structures into the passed array, popping them from the event_buf_t's internal buffer in a first-in-first-out order.

event_push() takes in a pointer to an array of events and an integer marking their amount. It pushes the events to the event buffer. However, the event buffer is conceptually a blocking queue. This means that if the event buffer is full, the event_push() function will sleep until the

thread sleeping on event_wait() signals it about new space having opened up. The design was chosen so that the server could not run out of memory due to too many events ending up in the buffer.

Sleeping until new events arrive or when more space opens up in a buffer was implemented using condition variables. Below are the implementations of the two functions.

```
void event_push(event_buf_t *buf, void *evs, int32 num) {
    int32 num_pushed = 0;
    int32 num_to_push;
    while(num_pushed < num) {
        mutex_lock(&buf->write_mutex);
        while(buf->num_events == buf->max_events)
            cond_var_wait(&buf->write_cond_var,
                &buf->write_mutex);
        num_to_push = MIN(buf->max_events - buf->num_events,
            num - num_pushed);
        memcpy((uint8*)buf->events + buf->num_events *
            buf->item_size,
            (uint8*)evs + num_pushed * buf->item_size,
            num_to_push * buf->item_size);
        buf->num_events += num_to_push;
        num_pushed += num_to_push;
        mutex_unlock(&buf->write_mutex);
        cond_var_signal_all(&buf->read_cond_var);
    }
}

int event_wait(event_buf_t *buf, void *evs, int32 max,
    int timeout_ms) {
    mutex_lock(&buf->write_mutex);
    while (!buf->num_events)
        if (cond_var_timed_wait(&buf->read_cond_var,
            &buf->write_mutex, timeout_ms)) {
            mutex_unlock(&buf->write_mutex);
            return 0;
        }
    int num = MIN(buf->num_events, max);
    memcpy(evs, buf->events, num * buf->item_size);
    memmove(buf->events,
        (uint8*)buf->events + num * buf->item_size,
```

```
        (buf->num_events - num) * buf->item_size);
    buf->num_events -= num;
    mutex_unlock(&buf->write_mutex);
    cond_var_signal_all(&buf->write_cond_var);
    return num;
}
```

As can be seen, events in the event buffer are stored in a contiguous array (event_buf_t's member events). Performance for larger buffers could potentially be increased by using a ring-buffer instead, as in that case there would be no need to move the remaining event structures when some of the front ones get popped.

3.4.3 An Example of Posting and Handling an Event

The proxy server has a separate thread for monitoring client –related network events. In the sections below the thread will be referred to as the *client thread*. The thread sleeps on a netpoll_wait() call until either a new client connection arrives, a client sends in some data, or something else regarding a client's connection happens.

If the netpoll notifies the application of a read event on a socket set in listening mode, a new connection is about to arrive. In this case the client thread calls the accept() function on the given listening socket, which returns a new socket for the incoming connection if successful, as well as the IP address of the client (The Open Group, 2017).

If the accept() call returns a valid socket, the client thread writes the socket file descriptor and the IP address into an event and pushes the event to the main thread's event buffer. Below is a pseudo-codified version of the client thread's main loop with procedure part visible.

```
for (;;) {
    netpoll_event_t events[64];
    int num_events = netpoll_wait(&_netpoll, events, 64, 5000);
    for (int i = 0; i < num_events; ++i) {
        e = &events[i];
        if (!(e->events & (NETPOLL_READ | NETPOLL_HUP)))
            continue;
        // The value 0xFFFFFFFFFFFFFFFF marks the listening socket
```

```
        if (e->data.u64 == 0xFFFFFFFFFFFFFFFF) {
            addr_t      a;
            socket_t    s = net_accept(_listen_socket, &a);
            if (s == KSYS_INVALID_SOCKET)
                continue; // Ignore invalid socket...
            event_t new_event;
            new_event.type                  = EVENT_ACCEPT_CLIENT;
            new_event.accept_client.socket  = s;
            new_event.accept_client.address = a;
            // The event_push call will sleep if no space is left
            event_push(com_event_buf, &new_event, 1);
        } else
            ...
    }
}
```

When the event_push() function is called from the client thread, the main thread, if currently sleeping on event_wait(), wakes up. It will check the type of the posted event and call a handler function based on the type. In the case of an EVENT_ACCEPT_CLIENT, it calls the function cl_accept() with a pointer to the event as a parameter. Most of the time this function allocates a stateful data structure for the connection. Otherwise it closes the connection if the proxy server currently has too many active connections, or if the IP address is not in a list of allowed IP addresses. This is an example of why stateful logic is executed on a single thread: allocating client data structures from a pool or checking allowed IP addresses from a table would otherwise require many locks or other synchronization methods to be thread-safe. Posting to the event buffer still requires locks, but they never need to be thought about afterwards. Below is a somewhat pseudo-codified example of what the main thread’s event loop looks like.

```
for (;;) {
    int num_events = event_wait(com_event_buf, events, 64, -1);
    event_t *e;
    for (int i = 0; i < num_events; ++i) {
        e = &events[i];
        switch (e->type) {
        case EVENT_ACCEPT_CLIENT:
            cl_accept(&e->accept_client);
            break;
        ...
        }
```

```
    }
}
```

3.4.4 Posting Variable Size Events

The size of the data of some events, such as network messages, may vary greatly. For this purpose, the proxy uses a general-purpose allocator with a segregated-fit type of scheme.

A segregated fit allocator stores free memory blocks grouped into lists by their sizes (Erin, 2001). In this case, any allocations are always rounded up to the nearest power of two (upwards). Using the nearest power of two, the highest bit can be found, and the index of that bit may be used as an index into an array of free-lists. This makes for constant-time lookups of free memory blocks of any size (Conte, 2016).

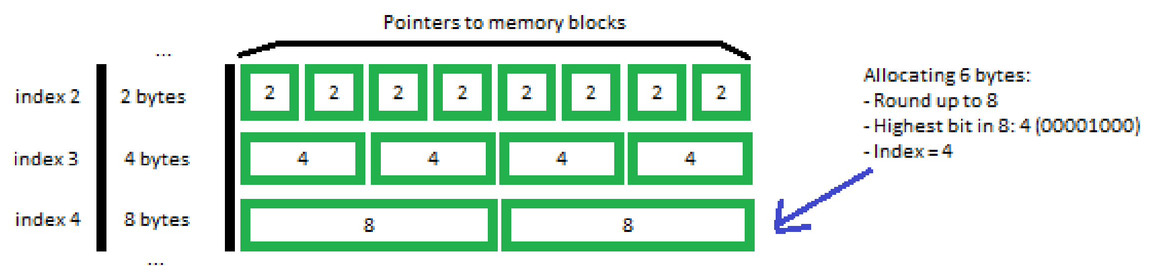


Image 6: Segregated fit allocation with the proxy's allocator

The allocator API, named *segpool*, looks as follows.

```
typedef struct segpool_t segpool_t;
void segpool_init(segpool_t *pool);
void segpool_destroy(segpool_t *pool);
void *segpool_malloc(segpool_t *pool, uint32 num_bytes);
void segpool_free(segpool_t *pool, void *ptr);
```

The proxy program has a single global segpool instance. A mutex is used to protect malloc and free calls to it. This prevents multiple threads from meddling with the context's state at the same time.

When a message from a client arrives, space for it is allocated from the segpool on the client thread. The message is copied and a pointer to the allocated memory is placed in the event structure that gets posted to the main thread. The main thread's event handler function is responsible for freeing the memory back to the segpool.

Although the segpool dynamically allocates memory as required in the background, the blocking design of the application's event buffer prevents malicious clients from sending in enough messages to make the proxy server run out of memory.

3.5 Modules

The proxy code is split into several modules. Each module is contained in its own source files with a single header file exposing the public API. Some of the modules have interdependencies, such as one module's function being called from another. There are no technical reasons to split the code like this, it is simply done for organizational purposes.

The modules are the following (with their prefixes in brackets): common (com), client (cl) and shard (shard). External functions belonging to any one of the modules are prefixed with the module's abbreviation.

3.5.1 The Common Module

The common module is the smallest one of the three. It contains common data and variables used by all of the rest of the program, such as the configuration settings and the main thread's event buffer, which are declared as global variables. The module has two public functions: com_init() and com_destroy(), which are called at the time of program initialization and shutdown respectively.

3.5.2 The Client Module

The client module contains the logic that handles client connections. The prefix used by the module is cl. The header file exposes 8 functions, the most important ones of which are listed below.

```
int cl_start();
```

cl_start() starts the module's thread and makes it listen for new connections. Called at start-up.

```
void cl_flush();
```

cl_flush() flushes packets that have been queued up to clients. Packets are buffered up so that less calls to the send() function are required.

```
void cl_accept(accept_client_event_t *event);
void cl_read(read_client_event_t *event);
```

cl_accept() and cl_read() are called from the main thread to handle accept and read events pushed to it by the module's thread.

```
void cl_disconnect(uint32 socket_index);
```

cl_disconnect() is called from the shards module when a shard asks the proxy to disconnect a client.

```
void cl_forward_shard_packet(uint32 client_index,
    const void *memory, int num_bytes);
```

cl_forward_shard_packet() is called from the shards module to forward a packet sent by a shard to a client.

When cl_start() is called, a new thread is started. The main loop of the module runs on this thread. The thread sleeps on netpoll_wait() until new connections come in or existing clients send messages. These events are posted to the main thread's event buffer.

Because the server requires stateful connections, each connection is stored in a data structure. The struct that represents a single accepted connection is the client_t.

```
enum client_flag {
    CL_FLAG_IN_USE      = (1 << 0),
    CL_FLAG_WILL_FLUSH  = (1 << 1),
    CL_FLAG_AUTHED      = (1 << 2)
};

struct client_t {
    socket_t    socket;
    uint32      id;
```

```
    uint32      flags;
    uint32      shard;
    uint32      flush_index;
    uint32      client_info;
    struct {
        int     num_bytes;
        uint8   memory[CL_IN_BUF_SZ];
    } in_buf;
    struct {
        int     num_bytes;
        uint8   memory[CL_OUT_BUF_SZ];
    } out_buf;
    cryptchan_t cryptchan;
};
```

Clients are stored in a fixed size pool with a free-list. No new client slots get allocated dynamically at runtime.

A client's ID, stored in the id member variable, is a unique 32 bit number. The number is used to identify a client in asynchronous calls, since a client structure may be re-used during the time it takes to complete an asynchronous call such as receiving from the client on the client network thread and processing the received data on the main thread. The CL_FLAG_IN_USE is used for a similar purpose – it gets set on when a client is created and set off when a client is disconnected.

The flush_index variable and the CL_FLAG_WILL_FLUSH flag are used in queuing the client up for a message flush when an outgoing message is queued up for it. If the CL_FLAG_WILL_FLUSH is not set, the client gets queued into an internal array of clients that should be flushed at the end of the current iteration of the main event loop. Outgoing messages are written to the out_buf variable.

The in_buf variable is used to store incomplete messages sent by the client that are waiting for more data to arrive.

The client_info variable is an index into a client_info structure inside the shard module. This index is only relevant if the CL_FLAG_AUTHED has been set.

The cryptchan variable contains encryption keys required to encrypt and decrypt messages. Cryptchan structures are used all throughout the MUTA codebase.

3.5.3 The Shard Module

The shard module is home to procedures dealing with connections to shards. It contains 10 public functions, some of which are listed below.

```
int shard_start();
```

shard_start() starts up the internal networking thread of the module, making it connect to shards listed in the proxy's configuration file and listen to messages from them. Called at program start-up.

```
void shard_read(read_shard_event_t *event);
```

shard_read() is called in the proxy's main event loop when a read_shard_event occurs. The event is posted by the shard module when a message is received from any connected shard over the network.

```
void shard_flush();
```

shard_flush() will flush out any messages currently queued to shards. Like for clients, shard messages are buffered and sent in groups.

```
bool32 shard_check_client_ip(uint32 ip);
```

shard_check_client_ip() is called from inside the client module. The function checks if an IP address, stored in a 32-bit integer, has been approved by a shard. When a player logs in, their IP address is sent to the shard server, which in turn sends it to the proxy server. Clients from unaccepted IP addresses are disconnected immediately.

```
int shard_check_client_auth_proof(const char *shard_name,
    const char *account_name, uint8 *token,
    cryptchan_t *cryptchan, uint32 client_index,
    uint32 *ret_shard_index, uint32 *ret_client_info_index);
```

Once a client has been connected and it has completed an encryption handshake, it must send an authentication proof to the proxy. The proof consists of an account name, a shard name and a 64-byte random token, generated by the login server. shard_check_client_auth_proof() is called from the client module when the authentication proof message is received. If the proof is incorrect, the client will be disconnected – otherwise, a message will be sent to the shard that the client has been accepted.

```
void shard_forward_client_packet(uint32 shard, uint32 socket_id,
    void *memory, int num_bytes);
```

shard_forward_client_packet() is called from the client module to forward a packet sent by a client to a shard.

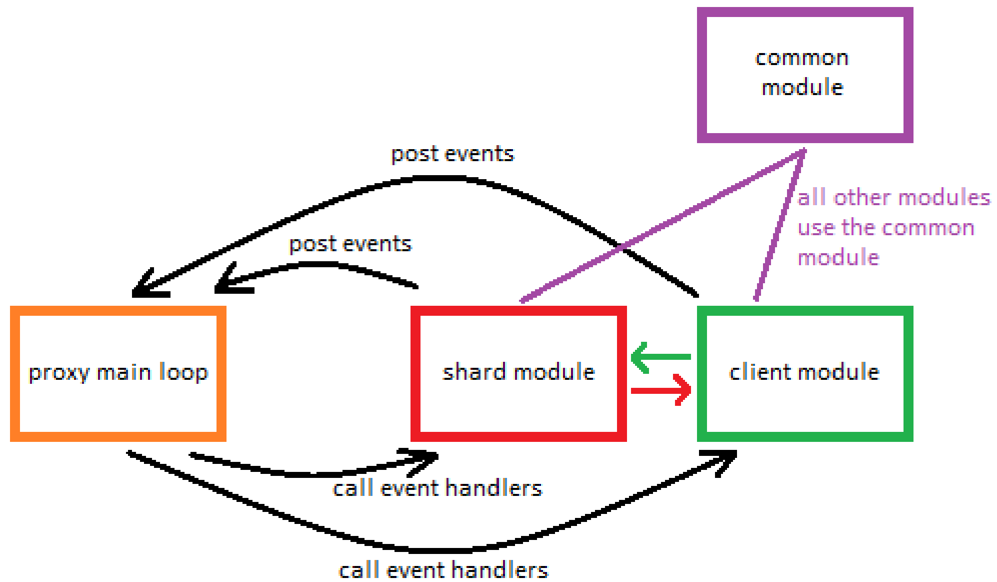
```
void shard_delete_client_info(uint32 shard, uint32 client_info,
    bool32 inform_shard);
```

shard_delete_client_info() is called from the client module when it decides to disconnect a client, and the client has been authenticated, meaning it has a client_info structure assigned to it. The function simply frees up the client_info structure. The inform_shard variable is used to tell the function if the shard the client was connected to should be informed of the disconnection. This flag is included because in some cases where the function will be called, the shard already knows the client will be disconnected.

3.5.4 Interactions between the Modules

The client and shard modules both use the common module's global data. For the most part, the shard and client modules both operate on their own threads from which they post events to the main thread. The application's main loop calls handlers from both of the modules to handle the events.

There is a slight dependency in between the two modules, in that each calls a single function from the other. It would have been possible to design this in a more decoupled way, but it was deemed a waste of time due to the program's small size and simplicity.



3.6 Converting the Shard Server to Work with Proxies

Originally, MUTA's client program connected directly to the shard server. For the purposes of rerouting messages through a proxy, the shard server had to see some changes.

3.6.1 The Proxy Module of the Shard Server

First a module that would handle connections from the shard to the proxy servers was implemented. The module is simply called *proxy*.

Proxy Sockets

The proxy module not only handles connections to proxy servers, but the indirect connections to clients as well. It keeps track of *proxy sockets*, data structures used to keep track of individual clients connected through a proxy. They are accessed via integer handles like network sockets, which is where their name comes from. Each proxy socket represents a single connection.

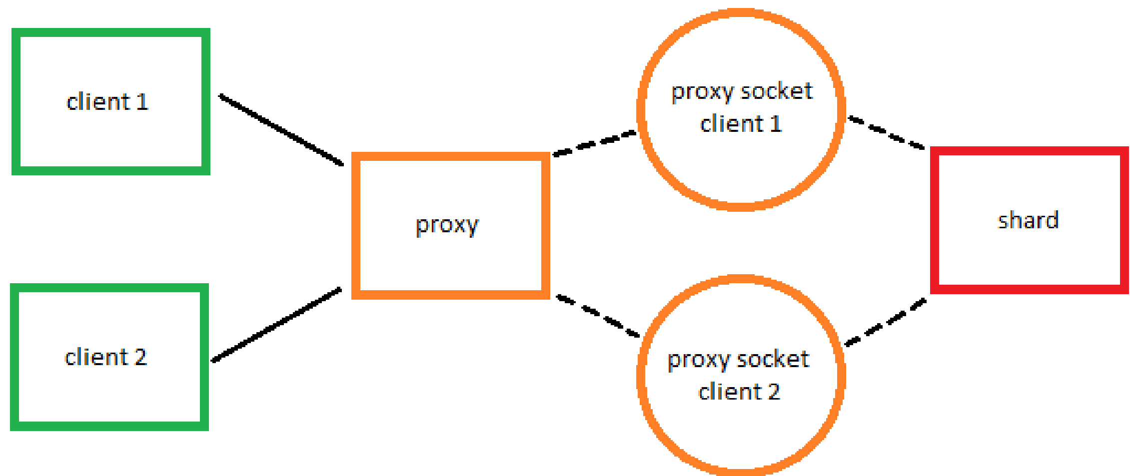


Image 7: A conceptual representation of how indirect client connections' traffic moves through proxy sockets between the proxy and the shard servers.

Each proxy socket gets a unique 32-bit integer identifier. Both, the proxy and the shard are made aware the identifier refers to a specific client.

Public Functions

The proxy module defines 11 public functions. The most relevant of them are defined below.

```
uint32 proxy_new_socket(const char *account_name, uint64 account_id,
    uint32 login_session_id, uint32 ip, uint8 *token);
```

proxy_new_socket() creates a new proxy socket for an incoming client. The return value is the handle to the proxy socket.

```
void proxy_cancel_socket(uint64 account_id, uint32 login_session_id);
void proxy_delete_socket(uint32 socket_index);
```

proxy_cancel_socket() and proxy_delete_socket() are called to close proxy sockets. The difference between the two is that proxy_cancel_socket() is used to cancel a yet-unconnected client's connection (that is, the login server has told the shard the client will connect soon, but it hasn't done so yet), whereas proxy_delete_socket() is used if the client has already connected.

```
bbuf_t proxy_forward_msg(uint32 proxy_socket, int size);
```

proxy_forward_msg() is used to forward a message to be sent to a client through a proxy server. Here the proxy socket handle is used. The return value is a bbuf_t structure, which is a buffer that can be used for writing messages to directly.

Implementing the Conversion

Any connection handling to clients was taken out of the shard program, as it would now only connect to proxy servers. Next, the implementations of the functions that are used to send data to clients were modified.

The shard code had three functions used to send messages to clients: cl_send(), cl_send_const_encrypted() and cl_send_var_encrypted(). The functions previously did some buffering, and at the end of each iteration of the shard server's main loop, the buffered messages would be sent to players. After the conversion, all these functions did was to call the proxy_forward_msg() function with a client's proxy socket index as a function argument. Hence, only these three functions needed to be changed in the shard's codebase for sending to work properly, plus each client's proxy socket index had to be stored in the client's state.

Reading client messages was moved to the proxy module. Before the conversion, it has used an event-based scheme where a networking thread posted events to the main thread.

3.7 Testing

After the server had been implemented, it was briefly tested using multiple machines in the same local area network. Five clients were used to connect through different proxy servers to a single shard. More testing at a larger scale is required in the future, but no problems were detected in these specific tests.

3.8 Proxy Server Behavior

This subchapter aims to explain at a higher level how the proxy routes packets between clients and shards.

3.8.1 Forming a Connections with Shard Servers

Each proxy has a configuration file that lists the IP addresses of shard servers it should forward traffic to and from. An entry in this file, named shards.cfg, looks like this.

```
shard = 127.0.0.1
```

If a shard cannot be reached or gets disconnected, the proxy will attempt to connect to it every two seconds. The blocking socket connect() calls are made on separate threads, which are joined after successful connection.

After successfully connecting to a shard, a cryptographic key exchange is performed. Once both sides have the correct cryptographic keys, they can send encrypted messages to each other. At this point, the proxy sends a message named FPROXY_MSG_LOGIN, which contains a password to the shard. Once the shard receives the message, it checks its legality, that is, if it is encrypted properly, and if the length of the packet is correct. Then the shard compares the password to its own password. If the passwords don't match, the proxy is disconnected. Otherwise the shard sends back a message named TPROXY_MSG_LOGIN_RESULT, which contains the shard's name and a result code.

3.8.2 The Proxy-Shard Protocol

A separate protocol is defined for communication between the shard and proxy servers. The protocol contains three different types of messages.

- PROXY_MSG messages can be sent by both, shard and proxy,
- TPROXY_MSG (from "to proxy") messages are sent by the shard to the proxy,
- FPROXY_MSG (from "from proxy") messages are sent by the proxy to the shard.

The protocol contains 11 message types.

1. PROXY_MSG_PUB_KEY contains a cryptographic public key at the beginning of a shard-proxy handshake,
2. PROXY_MSG_STREAM_HEADER contains a cryptographic stream header that will be exchanged during a shard-proxy handshake,
3. TPROXY_MSG_LOGIN_RESULT is sent to a proxy upon successful login to a shard,

4. TPROXY_MSG_OPENED_SOCKET is sent by a shard when it receives a message from a login server of an incoming client. It opens a proxy socket for the client. The message contains authentication data based on which the proxy should accept the client,
5. TPROXY_MSG_CLOSED_SOCKET is sent by a shard when it wants to disconnect a player. It contains the proxy socket ID of the player.
6. TPROXY_MSG_PLAYER_PACKET is sent by the shard when it wants to forward a message to a player client. It contains the client's proxy socket ID and the data to be sent.
7. FPROXY_MSG_LOGIN is sent by the proxy when it wants to log in to a shard. It contains a password for the shard.
8. FPROXY_MSG_OPENED_SOCKET_RESULT is a reply from the socket to a shard's FPROXY_MSG_OPENED_SOCKET message. It tells the shard whether the proxy was able to allocate a slot for the incoming client or not.
9. FPROXY_MSG_PLAYER_CRYPTCHAN is sent by the proxy to the shard after a client has successfully connected. It contains the cryptographic keys used for the proxy-to-client connection's encryption.
10. FRPOXY_MSG_PLAYER_PACKET is sent by the proxy to the shard when a packet is to be forwarded from the player to the shard. It contains the player's proxy socket ID and the data to be forwarded.
11. FPROXY_MSG_CLOSE_SOCKET is a command from the proxy server to the shard to close a proxy socket, typically if the client has sent an illegal message filtered by the proxy, or the client has disconnected.

3.8.3 Forming Connections with Player Clients

Before a client attempts to connect to a proxy server, it needs an authentication token, acquired from the MUTA login server. The player must log in to the login server with the correct user name and password, then choose a shard to connect to. After the player has chosen a shard, the login server generates a random 64-byte key, the authentication token. It sends the client's account name, IP address and authentication key to a shard server, which in turns forwards them to a proxy server if successful. The shard replies to the login server with a message that describes if the player can connect to the shard or not. The shard is responsible of choosing a proxy server through which the client's connection will be routed. The IP address of the proxy server is also sent in the case of success.

If the player is allowed to join by the shard, the login server sends the player the authentication key and the IP address of the chosen proxy.

Once the player receives the proxy's IP address and authentication key, it disconnects from the login server and opens a connection to the proxy server. The proxy server has a table of IP addresses accepted by shards: if the IP of the player is not in the list, the client will be disconnected immediately.

Once connected, the client and the proxy exchange cryptographic keys, after which the client must send the authentication proof: authentication token and the account name. If the proxy does not find the key and the account name in the list of allowed clients, or the key and name don't match, it will disconnect the client. Otherwise it tells the shard the client has connected. If the shard responds with success, any messages from the client will now be forwarded to the shard, and any messages appointed to the client by the shard will be sent to the client.

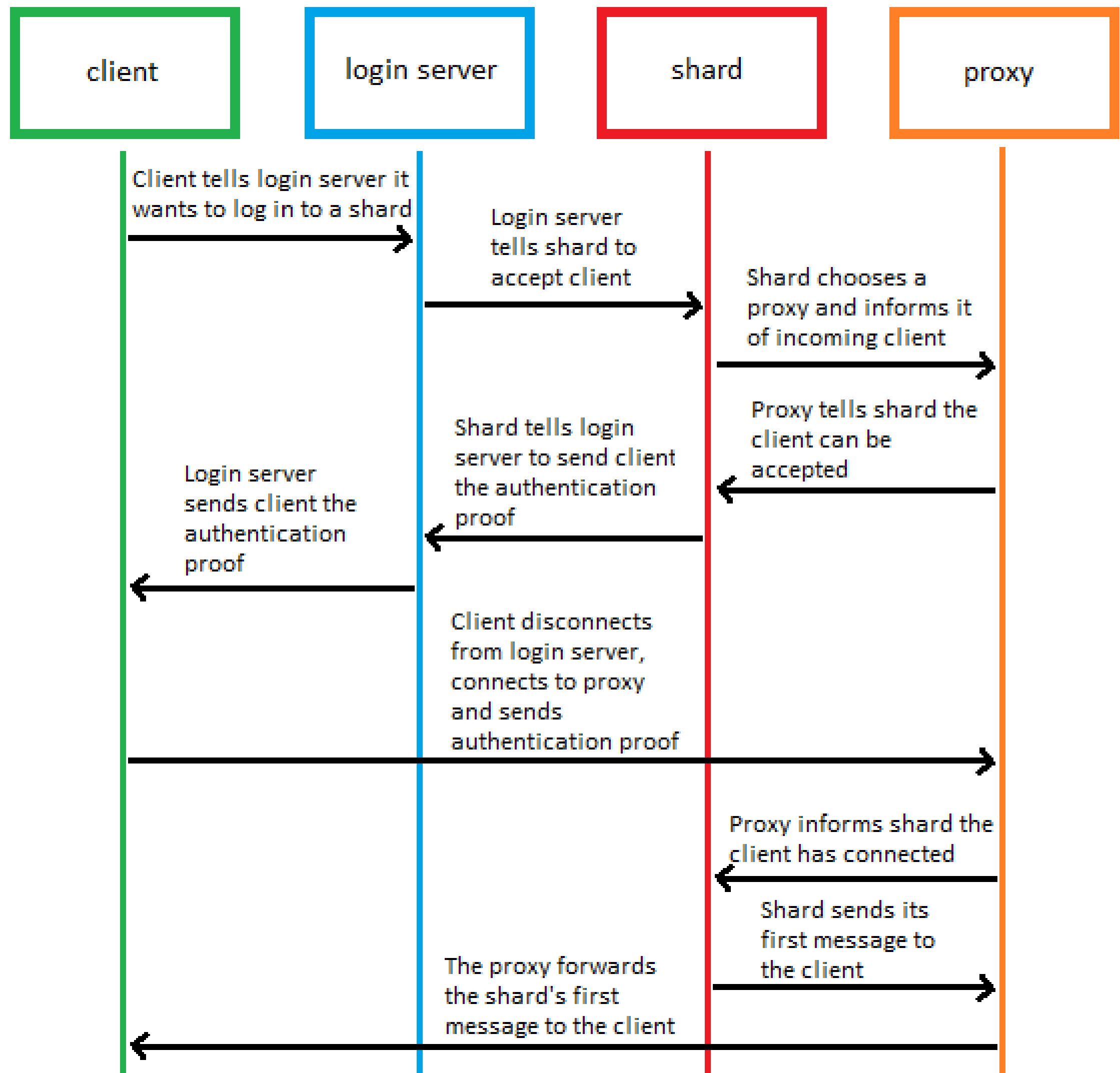


Image 8: Forming a client-shard connection through the proxy

3.8.4 Routing Packets between Clients and Shards

When a shard is informed of an incoming player by the login server, it opens a new proxy socket that's used to keep track of the indirect connection's state. The socket is generated an unsigned 32-bit integer identifier, and the identifier is sent to the proxy server. Now both applications know the identifier refers to a specific client, so they can use it while communicating between each other.

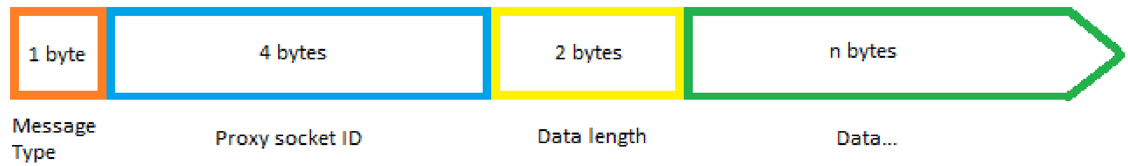


Image 9: Description of a packet that forwards a client’s message from a proxy to a shard. The client’s message is included in the data portion of the packet.

Packet Filtering

The proxy does not read the messages sent by the clients completely, but it does some primitive packet filtering. It checks if the encryption has been tampered with and if the sizes of messages are correct. For variable length messages, it checks all size variables as well. Hence, the proxy is somewhat aware of the client-shard protocol of MUTA.

If a client message is deemed illegal by the proxy, that is, its size does not match the size limits set by the protocol, the client is disconnected. Otherwise the message will get forwarded to the shard. The shard performs the exact same size checks as the proxy just in case, but also other, packet-specific checks as it might use the packet to manipulate game state.

3.8.5 Further Development

The practical part of writing the proxy server for this thesis was completed mostly in less than a month, and at the same time, MUTA’s login server was also implemented. Such a short period of development time was hardly enough to implement all of the desired features of the proxy server system.

Timing Clients Out

More timers are planned across the infrastructure to make requests and clients time out in case of one server being unable to respond to a message. This would prevent clients from being unable to log in for short periods of time when unexpected events such as application crashes happen.

Improving the Shard-Proxy Protocol

The shard and proxy servers are to be made communicate more about their configurations, such as how many clients should each allow to connect at maximum capacity, etc.

Working with Changing IP Addresses

Under some circumstances, it might be a client's IP address changes between the connection swap from the login server to the proxy server. Currently we expect both connections to arrive from the same IP address. It is planned the IP address would no longer get checked upon a connection swap.

More Thorough Testing

More testing needs to be done to confirm the proxy server scales along with an increasing user count, and that it has no hidden bugs.

4 Conclusions

In this thesis, a proxy server for the in-development MMORPG MUTA was implemented.

The proxy acts as a link between MUTA player clients and MUTA shards. It does primitive packet filtering in order to not pass messages deemed illegal as by the MUTA client-server protocol to the server by checking if the packets are of the right sizes for their types, if their data length fields appear correct, and if their encryption has not been tampered with.

The proxy can be used to hide the IP addresses of MUTA shards from clients. It also allows for a scheme where multiple players connect to the same game world through different IP addresses, since multiple proxies can be connected to a single shard simultaneously.

In conclusion, while the server has not been thoroughly field-tested yet with high amounts of players, for low amounts of players it appears to work correctly in that it is capable of redirecting game traffic as well as filtering it.

Sources

Bartle, Richard. (2003). Designing Virtual Worlds, Introduction to Virtual Worlds (pages 4-5). San Francisco, USA: New Riders Press.

MPOGD (2007). [Tibia Celebrates 10th Birthday]. MPOGD. From address https://web.archive.org/web/20150402110535/http://www.mpogd.com/news/?ID=2316

Andrew Josey. (2017). [POSIX® 1003.1 Frequently Asked Questions (FAQ Version 1.16)]. The Open Group. From address http://www.opengroup.org/austin/papers/posix_faq.html

The Open Group. (2017). [accept – accept a new connection on a socket]. The Open Group. From address http://pubs.opengroup.org/onlinepubs/9699919799/functions/accept.html

Conte, Matt. (2016). [tlsf]. From address https://github.com/mattconte/tlsf/blob/master/README.md

Erin. (2001). [Memory Allocators]. From address https://www.cs.nmsu.edu/~ekerriga/presentation/index2.html

Denis, Frank. (2017). Libsodium documentation. From address https://download.libsodium.org/doc/secret-key_cryptography

Rescorla, E. (1999). Diffie-Hellman Key Agreement Method. From address https://tools.ietf.org/html/rfc2631

Microsoft. (2018). WSAPoll function. From address https://docs.microsoft.com/en-us/windows/desktop/api/winsock2/nf-winsock2-wsapoll

CCP Games. (23.5.2016). EVE Online's Citadel Expansion Builds Dreams and Wrecks Them. From address https://www.ccpgames.com/news/2016/eve-onlines-citadel-expansion-builds-dreams-and-wrecks-them/

FarmerJ03. (27.1.2014). Reply to post Interested in MMO server architecture. From address https://www.reddit.com/r/gamedev/comments/1w746u/interested_in_mmo_server_architecture/

Carlin, Chad. (26.10.2006). WSAPoll, A new Winsock API to simplify porting poll() applications to Winsock. From address https://blogs.msdn.microsoft.com/wndp/2006/10/26/wsapoll-a-new-winsock-api-to-simplify-porting-poll-applications-to-winsock/

George. (20.1.2014). Practical difference between epoll and Windows IO Completion Ports (IOCP). From address https://www.ulduzsoft.com/2014/01/practical-difference-between-epoll-and-windows-io-completion-ports-iocp/

Koster, Raph. (2009). Database “sharding” came from UO? From address https://www.raphkoster.com/2009/01/08/database-sharding-came-from-uo

Payer, Stephan. (2012). Cheaters, Hackers and Script Kiddies – The Dark Side of Online Games. From address https://gdcvault.com/play/1016531/Cheaters-Hackers-Script-Kiddies-The

Dunn, Fletcher. (2018). Denial-of-Service Mitigation. From address https://gdcvault.com/play/1024933/Denial-of-Service

Schubert, Damion. (2012). Classic Game Postmortem: Meridian 59. From address https://www.gdcvault.com/play/1016637/Classic-Game-Postmortem-Meridian

Rudy, Matthias. (2011). Inside Tibia – The Technical Infrastructure of an MMORPG. From address https://gdcvault.com/play/1014908/Inside-Tibia-The-Technical-Infrastructure

Springer, Rink. (2015). How hackers grind an MMORPG: by taking it apart! From address https://www.youtube.com/watch?v=9pCb0vIp_kg

Wyatt, Patrick. (2012). Writing Server and Network Code for Your Online Game. From address https://gdcvault.com/play/1015609/Writing-Server-and-Network-Code