FACULTY OF FUNDAMENTAL PROBLEMS OF TECHNOLOGY
WROCŁAW UNIVERSITY OF TECHNOLOGY

# NETWORK ENGINE FOR INTERACTIVE REAL-TIME SIMULATIONS

BARTOSZ CHODOROWSKI

B.Sc. Thesis written
under the supervision of
Marcin Zawada, Ph.D.

WROCŁAW 2011

# Abstract

The goal of this work is to design and implement a library (software engine) which would be responsible for network communication between programs that involve real-time motion of multiple objects, such as physics simulations, computer games.

Transmitting and receiving data, which describe motion of objects, through the Internet may be a tough challenge. Typical problems include various network latencies as well as unreliability of communication channels. However, by using certain algorithms, it is possible to make clients feel as if they were a part of one, consistent physics simulation, even though they can be spread around the world.

## Acknowledgements

I sincerely would like to thank Kacper Paszke and William E. Minsker for the effort they put in checking the language of this work.

# Contents

# 1 Introduction

Internet access is getting faster and cheaper every day. However, network latencies are inevitable and still are (and probably will be for a long time) a major problem when it comes to transferring data in order to provide a real-time interactive physics simulation.

Generally, programmers who work on networked simulations (or action games) have to deal with network lags on their own. In this work Bazinga is presented — a network engine for the interactive real-time simulations. Using this piece of software, programmers can easily focus on physical aspects of the simulation and not to concern about network issues whatsoever.

A synchronization of physics between sides who participate in simulation (hereinafter "clients") and side which manages network connections of participants (hereinafter "server") may be done in a few ways. This work describes the most notable approaches. They are also implemented, so one can switch between synchronization methods to see the differences and choose between them according to the current needs.

Compression is also useful. *zlib* [7] is used in order to compress every big network packet. *Delta compression* is also supported. Furthermore, lossy compression of floating-point numbers has been implemented so that programmers can reduce the precision when it is not essential.

Section 2 provides an introduction into the subject. In section 3, the main theoretical issues will be presented in details. In section 4, Bazinga project will be designed from scratch. Section 5 covers its implementation and points out the most interesting practical issues. Finally, section 6 provides conclusions.

## 1.1 Goals

Goals of this work are defined as follows:

- discuss the methods and problems related to effective network synchronization of multiple objects
- design reusable library which facilitates creating network real-time applications
- offer efficient implementation
- provide easy to understand examples of usage

## 1.2 Tools

The *C++* language (see [12]) has been chosen for the implementation as it is a compromise solution between the need of some high-level features and performance. Not only is it convenient to use Object-Oriented Programming paradigm (OOP), but it is also helpful in achieving clear code and making its maintenance easier. It is natural to use this paradigm when it comes to representing physical objects in virtual reality — this is the reason why OOP is so commonly used in game programming.

*Simple DirectMedia Layer* (*SDL*) is a cross-platform multimedia library designed to provide low level access to audio, keyboard, mouse, joystick, 3D hardware via OpenGL, and 2D video framebuffer [1]. Yet Bazinga engine does not handle rendering or multimedia at all — *SDL* is needed merely to handle threads and delays in a cross-platform way. It works with *C* and *C++*, so it is perfect for this project.

*Berkley Sockets API* is used to provide high performance in communication via *UDP/IP* protocol. *WinSock 2* is an API which is almost completely compatible with *Berkley Sockets*. It allows to use such mechanisms in *Windows* operating systems. See [4] for complete reference to socket programming.

*Blossom Math* [11] is a library which includes fair implementation of mathematical entities such as vectors, matrices, planes. It is a part of the *Blossom Engine* — a set of libraries for game developers. Bazinga uses its `vec3` class to store physical properties of objects.

### 1.3 Features

Compact list of Bazinga features includes:

- real-time synchronization of multiple objects
- each object consists of some physical properties and multi-purpose binary buffer
- events (possibility of reliable delivering through unreliable communication channel)
- physics rewinding algorithm
- capability of sending neighbouring objects only
- clients' data packets consist of bits that denote pressed keys
- scalable floating point precision
- relativity of transferred positions
- lossless compression, Delta Compression
- connections management
- lag information

## 2 Background

Basic information, essential to handle network programming, has to be discussed first. One might wish to proceed to section 3 if he or she is familiar with the basics of programming network applications and computer networking in general.

### 2.1 Communication Overview

Interactive network simulation is a kind of distributed application — a number of programs need to communicate over a computer network in a specific way (using an a priori defined protocol) to provide sensation that clients participate in one virtual world. It is assumed that clients use the Internet in order to transmit and receive data.

The most common way to organize communication is to use *Client-Server Model*. That is to say, one program needs to be dedicated to the server. Each client has to establish a network connection

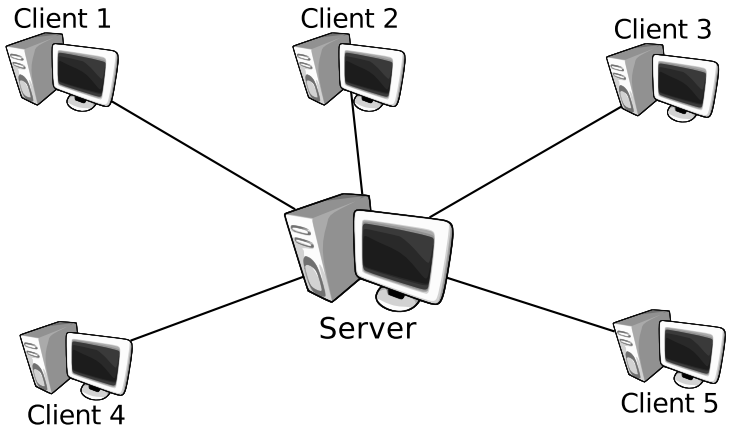


**Figure 1:** Client-Server Architecture

with it, thus the server is the centre of this architecture. Client-Server Model is presented in the Figure 1. Other forms of organizing communication are rather impractical to meet the needs described in the previous paragraph.

Another concept, every network programmer must be aware of, is the *Open Systems Interconnection model (OSI model)* [6]. It divides the whole process of intercommunication into seven layers. Each of them is a collection of similar functions (provides some functionality) and communicates with one layer above and one below. The theoretical OSI model is presented in the Figure 2.

Generally, the OSI model does not apply in practice in a strict manner — many protocols can be precisely associated with some layers, others provide functionality of several layers (e.g. TCP is generally assigned to the transport layer while it is also responsible for a graceful close of sessions which, regarding to the OSI, should be the duty of the session layer). Some layers may be completely neglected — lots of communication takes place without compression or encryption at all (it is a domain of the presentation layer). In spite of that, the OSI is a good model to understand how network protocols should be designed, how they should function and communicate with each other in theory.

Usually, network applications programmer is not concerned about 1-3 layers. They are either related to hardware or completely handled by the operating system. Using Berkley Sockets, the programmer can choose transport layer protocol (that is TCP or UDP) and build his or her own application protocol on top of it. That protocol can obviously provide functionality known from layers 5 and 6 of the OSI model (and so Bazinga network protocol offers compression).

## 2.2 TCP vs. UDP

The *Transmission Control Protocol (TCP)* allows exchanging data between two programs in a reliable way. The TCP allows clients to establish connection with a server and it ensures that data is delivered exactly in the same order that it was sent. Every piece of transmitted information has to be acknowledged by the recipient.

Alternative to TCP is *User Datagram Protocol (UDP)*. It is a much more simple protocol. It allows the sending of independent packets called *datagrams*. Unlike TCP, UDP does not provide

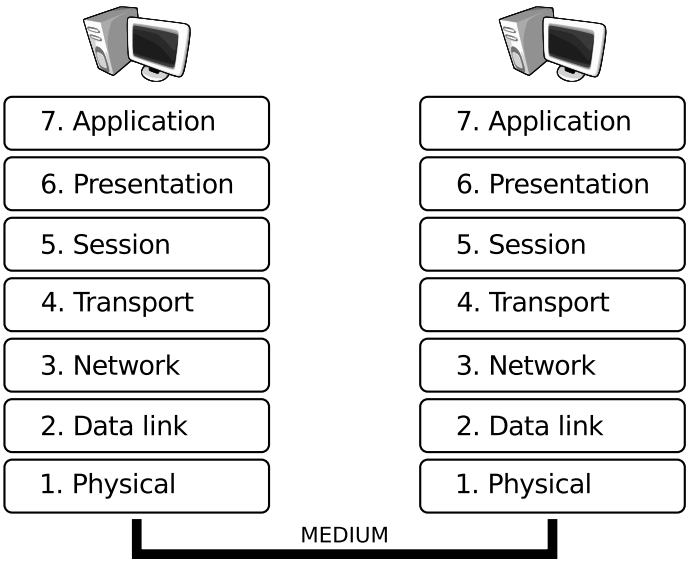


**Figure 2:** ISO model

reliability. Some datagrams may be lost, duplicated or received in a wrong order. However, it carries checksum with every datagram, so it is very unlikely that data will be delivered across the network malformed — it will either reach its destination or be lost. UDP also does not provide connection mechanisms known from TCP — datagrams may be sent and received without a connecting procedure.

Advantages of TCP are indisputable. Nevertheless, the price that is paid for its features may be overwhelming — efficiency. In TCP everything must be acknowledged. That generates more network traffic. What is more important, when one packet is lost or reaches its destination in a significantly longer time, other urgently needed data may have to wait. Those reasons almost always preclude usage of TCP in real-time applications. Physics simulations mostly can afford some level of packet loss, whereas awaiting for stray network packets or demand of retransmission is unacceptable.

# 3 Problem Analysis

Although data exchange in networked physics simulation may not seem to be a difficult task, there are some issues that every network programmer encounters. This section discusses them.

## 3.1 Thin Server Approach

The main loop of standard non-networked physics simulation is shown in Algorithm 1. `now()` function returns current time in high resolution. `processInput()` serves keyboard and mouse events. `physics(`$t_{i-1}$`,` $t_i$`)` performs update of physical attributes of each object (such as position, velocity, etc.) from time $t_{i-1}$ to time $t_i$. `render()` draws everything that is needed to be seen on the screen.

```
1 t_0 := now();
2 for i := 1 to ∞ do
3 |   t_i := now();
4 |   processInput();
5 |   physics(t_{i-1}, t_i);
6 |   render();
```

**Algorithm 1:** Main loop of non-networked physics simulation

Basically, the idea of Thin Server is as follows: every client deals only with physics of one object it controls (e.g. it applies force to a specified object when proper keys are pressed on the keyboard). After each physics update it sends information to the server about physical attributes of its object. The server broadcasts this information unchanged to other clients, who update the attributes in their simulation.

The Thin Server method makes an assumption that each client has one assigned object and its interactions hardly ever influence other objects. This assumption is reasonable when considering e.g. flight simulators — planes scarcely ever collide with each other and if they do, the effect is destructive and behaviour of planes wreckages after collision is not so important. First person shooter games with a static environment also may meet this limitation. After all, players who shoot each other do not collide with others too often. On the other hand, we can imagine a computer simulation which allows each client to apply force to arbitrarily chosen object. The Thin Server method cannot be applied in such situation.

It is worth mentioning that dividing physics at each client may be a huge efficiency gain. Every client is concerned about its own object only and it does not have to waste time to compute other objects' physical attributes.

A major disadvantage of this approach is the possibility of cheating. There is no control over the data that a client transmits. If the client with bad intentions modified an application code, it could arbitrarily change its object's attributes, which is often unacceptable (imagine a situation where an evil flight simulation cheater can freely change his plane's position and velocity regardless of the law of physics). This issue usually forces a completely different approach.

## 3.2 Thin Client Approach

To avoid cheating, the server has to become authoritative over physics. An exciting idea comes to mind — let us make the client a thin terminal, so its responsibilities include only two tasks. First is handling input and sending information about it to the server. Second is receiving data about the objects from the server and render the scene on the screen. Thereby, only the server is responsible for the physics. There is no chance for cheating, as it is known from the Thin Server Approach. It is important to make sure that the server keeps up with computing the physics, for that may be the bottleneck.

Packets clients sent in this method usually comprise bits denoting whether specified keys on the keyboard are pressed at the moment. Simulations that use a mouse must also send information about the mouse movement. Packets that go the other way consist of description of objects (physical attributes like position and velocity are the most important, yet other properties such as colour or shape may be sent as well).

Both client and server can work multi-threaded. On the server side, one thread can compute physics and transmit update data at fixed time intervals, while a second thread receives data from

clients. On the client side, one thread can handle keyboard events and transmit data, while the second receives data from the server and renders the virtual world. In practice, fixed time intervals between transmitted packets are good (yet they have to be chosen, depending on the specific needs).

## 3.3 Physics at Client's Side

Thin Client may also be modified, so it is not so thin anymore — it can compute simplified physics on his own during the time intervals between updates. When a packet comes from the server, the client corrects its state. It is very convenient, because it makes receiving data rate and screen redraw rate independent (so that world can be rendered with more frames per second).

### 3.3.1 Smoothing

Irregular network latencies constitute another issue. Physics updates, that the server sends with fixed time rate, may be received by the client at irregular intervals. This may cause unexpected jumpy movement of objects at the client's side.

One of the possible solutions to this problem is application of smoothing algorithm. *Exponentially Smoothed Moving Average* (described, inter alia, in [13]) is very common and easy to implement, thus is described in this paper and implemented in Buzinga engine. When the client receives update of property $\bar{x}_k$ from the server, it applies weighted mean with current value $\bar{x}_{k-1}$ of property that is being corrected. A priori chosen constant $\alpha \in (0,1]$ is used as a weight:

$$\bar{x}_k := \alpha\bar{x}_k + (1-\alpha)\bar{x}_{k-1}$$

where property is usually up to three dimensional vector of position or velocity: $\bar{x} = [x_x, x_y, x_z]$.

As the updates come incessantly, a smoothing effect is achieved.

### 3.3.2 Physics Rewinding

One may be not satisfied with the smoothing algorithm as a solution to the problem. After all, it is just an attempt to deceive the senses, using smoothness for precision trade-off.

Knowing the frequency with witch the server sends updates with, the client can apply corrections in a more clever way. Updates have to be numbered, so the client can reject out of order packets and determine expected time of the next packet. If the update comes earlier than the client expected, it applies received data verbatim (so called "snap" occurs). Otherwise, the client computes `physics()` from expected receive time to the current time after it receives the packet (this can be interpreted as rewinding back in time to a designated point and recomputation of the physics). Wisely choosing expecting times of approaching updates, snapping can be significantly limited. However, it introduces some prediction of physics on the client's side that might go a little bit wrong — when the client is not aware of the significant input changes of other clients. This is the reason why it is a good idea to use a smoothing algorithm together with physics rewinding as they are completely orthogonal mechanisms.

```
1  (lastSeq, data) := waitForUpdate();
2  updateExpectedAt := now() + updatePeriod;
3  while true do
4      (seq, data) := waitForUpdate();
5      if seq ≤ lastSeq then
6          continue;
7      if seq ≠ lastSeq +1 then
8          updateExpectedAt := updateExpectedAt + updatePeriod · (seq − lastSeq −1);
9      lastSeq := seq;
10     applyDataFromUpdate(data);
11     if now() < updateExpectedAt then
12         updateExpectedAt := now() + updatePeriod;
13     else
14         physics(updateExpectedAt, now());
15         updateExpectedAt := updateExpectedAt + updatePeriod;
```

**Algorithm 2:** Receiving and applying corrections in physics rewinding method

Pseudo-code of physics rewinding method is shown in Algorithm 2. Note that `waitForUpdate()` function halts execution of the thread until it receives update from the server. It returns a sequence number of received update and the data that is to be applied with `applyDataFromUpdate()` function. `updatePeriod` is a constant determined beforehand with the server. Of course, handling input, sending data to the server, continued physics computation and rendering takes place in another thread.

## 3.4 Client Side Prediction

All previously presented mechanisms have a disadvantage. Namely, when a simulation participant presses keys on the keyboard, the effect is seen on the screen after some time which is equal to the network latency (time that packet needs to reach the server and come back).

The *Client Side Prediction* is a mechanism which resolves that issue. The basic idea is that the client reacts on its input immediately, predicting what is to be happening on the server. Nonetheless, assumption of "one client – one object" is needed again to perform this method as well as some major changes in both client's and server's architecture. This is the reason why when designing Bazinga project, it was decided not to include this mechanism.

Article [3] concisely covers this issue.

## 3.5 Compression

Server updates can describe quite a lot of objects. Unfortunately, the packet size is limited by the *Maximum Transmission Unit (MTU)*, which is imposed by the medium (e.g. MTU for Ethernet is 1500 bytes). IP and UDP headers have to be included too, so the limitation gets even more strict. Some protocols are very cautious — DNS uses UDP only if data length does not exceed 512 bytes (as defined in section 4.2.1 of [8]).

This is where compression comes to the rescue. General lossless compression of update packets is a common way to keep the size of packets low.

### 3.5.1 Delta-Compression

Analyzing the content of the update packets, it may be noted that subsequent packets are largely the same. Assuming that the client has received the last update, server can send only the differences between the last and current packet (so called *Delta-Compression*). This allows the reducing of the amount of transferred data, but on the other hand, makes the transmission less resistant to packet loss. Constant $n$ needs to be adjusted so that every $n$-th update goes without Delta-Compression mechanism (to resynchronize clients that has lost some packet).

Let $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_m)$ be sequences of bits. Let $\oplus$ be an operator described with the following table (it is a well-known XOR operator):

| $\oplus$ | **0** | **1** |
|---|---|---|
| **0** | 0 | 1 |
| **1** | 1 | 0 |

it may be also intuitively defined for sequences:

$$A \oplus B = \begin{cases} (a_1 \oplus b_1, a_2 \oplus b_2, \ldots, a_n \oplus b_n) & \text{if } n = m \\ (a_1 \oplus b_1, a_2 \oplus b_2, \ldots, a_n \oplus b_n, b_{n+1}, b_{n+2}, \ldots, b_m) & \text{if } n < m \\ (a_1 \oplus b_1, a_2 \oplus b_2, \ldots, a_m \oplus b_m, a_{m+1}, a_{m+2}, \ldots, a_n) & \text{if } n > m \end{cases}$$

To transmit the differences only, the server stores the last sent update $L = (l_1, l_2, \ldots, l_n)$ for each client. Instead of sending the whole current update $C = (c_1, c_2, \ldots, c_m)$, the server computes $L \oplus C$, compresses it and finally transmits. Note that since $l_i = c_i$ for many values of $i$ (due to assumption that update data has not changed much), entropy of $C \oplus L$ is low. Therefore compression is much more efficient.

### 3.5.2 Floating-Point Precision

Physical properties of objects are usually up to 3-dimensional vectors of real numbers. Computers represent real numbers, using floating point — *IEEE 754-2008* [5] is the most widely-used standard. In this subsection the main concepts of floating-point representation will be presented, so one can easily understand how to dispense the unnecessary precision to reduce the size of data that is needed to be transferred.

The basic idea is to express the number so it can be written according to the following formula:

$$a = (-1)^s \cdot c \cdot 2^q$$

where:

- $s$ – sign (two possible values: 1 denotes negative, 0 when positive)
- $c$ – significand (coefficient, mantissa), $c \in [1, 2)$
- $q$ – exponent

Storing the sign in a computer memory is not a problem at all — only one bit is needed.

Mantissa can be written as a binary fixed-point number. As it is normalized (from the $[1, 2)$ interval), its binary representation always starts with 1 followed by the decimal point. Therefore, there is no reason to waste bits in the memory to remember what is before the decimal point. Let's assume

that $c_n$ bits of the fractional part of significand are stored in the memory — the rest is irretrievably lost.

Exponent is an integer, which may be negative, positive or zero. $q_n$ bits are needed to encode $2^{q_n}$ different values. To avoid encoding of negative numbers, offset $\sigma$ is added to the exponent in order to store it in the memory. Besides, minimum and maximum possible values are reserved and handled specially. Therefore, minimum and maximum values of the exponent are:

$$\begin{aligned} q_{min} &= 1-\sigma \\ q_{max} &= 2^{q_n}-2-\sigma \end{aligned}$$

$\sigma$ is usually selected in a way that both big and small numbers are handled well. When $\sigma = 2^{q_n-1}-1$, then the value of the exponent is an integer from $[-(2^{q_n-1}-2),\ 2^{q_n-1}-1]$ interval.

When stored exponent is all zeros or all ones, interpretation of the number is special, so that 0, $\infty$, $-\infty$, `NaN` and subnormal numbers can be encoded. Let $M$ be a sequence of significand bits. Following table summarizes floating point representation of real numbers:

| $q+\sigma$ | $q$ | Value being represented |
|---|---|---|
| 0 | | $(-1)^s \cdot 2^{1-\sigma} \cdot 0.M$ |
| $1 \ldots 2^{q_n}-2$ | $1-\sigma \ldots 2^{q_n}-2-\sigma$ | $(-1)^s \cdot 2^q \cdot 1.M$ |
| $2^{q_n}-1$ | | $\pm\infty$ when $M=0$, `NaN` otherwise |

IEEE 754-2008 defines *binary32* format (`float` type known from C/C++) with following constants: $q_n = 8$, $c_n = 23$, $\sigma = 127$. Figure 3 shows an example of encoding a number in *binary32* floating-point standard.


**Figure 3:** Representation of 3.2 in *binary32* format

Knowing how real numbers are stored in the computer memory, the network programmer can write his own implementation of floating-point numbers. $q_n$, $c_n$, $\sigma$ constants can be arbitrary chosen to meet the current needs. Some specific floating point numbers can be packed into 16- or even 8-bit variables. Bazinga engine offers fair implementation where the programmer defines the aforementioned parameters.

# 4 Project

Bazinga engine has been designed as a reusable library for both client and server programs.

## 4.1 Class Diagram

Diagram of classes, which are important for the user, is shown in the Figure 4. Only public and the most significant attributes and operations have been included. `Client` and `Server` are singleton instances of `CClient` and `CServer` classes (it is not explicitly shown in the diagram).

## 4.2 API

Usage of Bazinga library is very simple. In the server program, the programmer has to run `CServer::create()` function in the first place. Its declaration is as follows:

```
bool CServer::create(
  int keysCount,
  void (*eventsProcessor)(const CEvent& e, ConnectionID id),
  bool (*doPhysics)(Time, Time),
  bool (*clientConnected)(CConnection* c),
  bool (*clientDisconnected)(CConnection* c));
```

`keysCount` is an important parameter. It must be the same at the server and all the clients. Other parameters are function pointers. `eventsProcessor` is called when event occurs. `doPhysics` is the physics function which performs the simulation step. `clientConnected` and `clientDisconnected` are called when any client joins or leaves the simulation. None of the function pointers described above can be `NULL`.

After the server is created, the programmer can set various properties, according to the current needs, using those functions:

```
bool CServer::setCompressionMethod(CompressionMethod::Type compressionMethod);
bool CServer::setMotdBuffer(const byte* motdBuf, uint motdBufLen);
bool CServer::enableRelativePositions();
bool CServer::setSendingNeighbouringObjectsOnly(float contigousOnly);
bool CServer::setUpdatePeriod(Time updatePeriod);
bool CServer::setClientsInformationPeriod(int clientsInformationPeriod);
bool CServer::setPrecision(const CPrecision& first, const CPrecision& second, const
     CPrecision& third, const CPrecision& fourth);
```

Then the server should call `Server::listen(Port port)` and execute the main loop, using `CServer::simulationLoop()`. When the simulation is over, the program returns from the function. The programmer should call `CServer::destroy()` in order to clean everything up.

Programming the client program is very similiar. `CClient::create()` is defined as:

```
CClient::create(
  int keysCount,
  void (*eventsProcessor)(const CEvent& e),
  bool (*doPhysics)(Time, Time),
  bool (*doRender)(),
  bool (*doInput)(),
  bool (*doLoadStuff)());
```

New function pointers are essential. `doInput`'s task is to handle all the keyboard and mouse events. The client might also want to load some data after connecting to the server — `doLoadStuff` function is called then.

Parameters setting is done using the following methods:

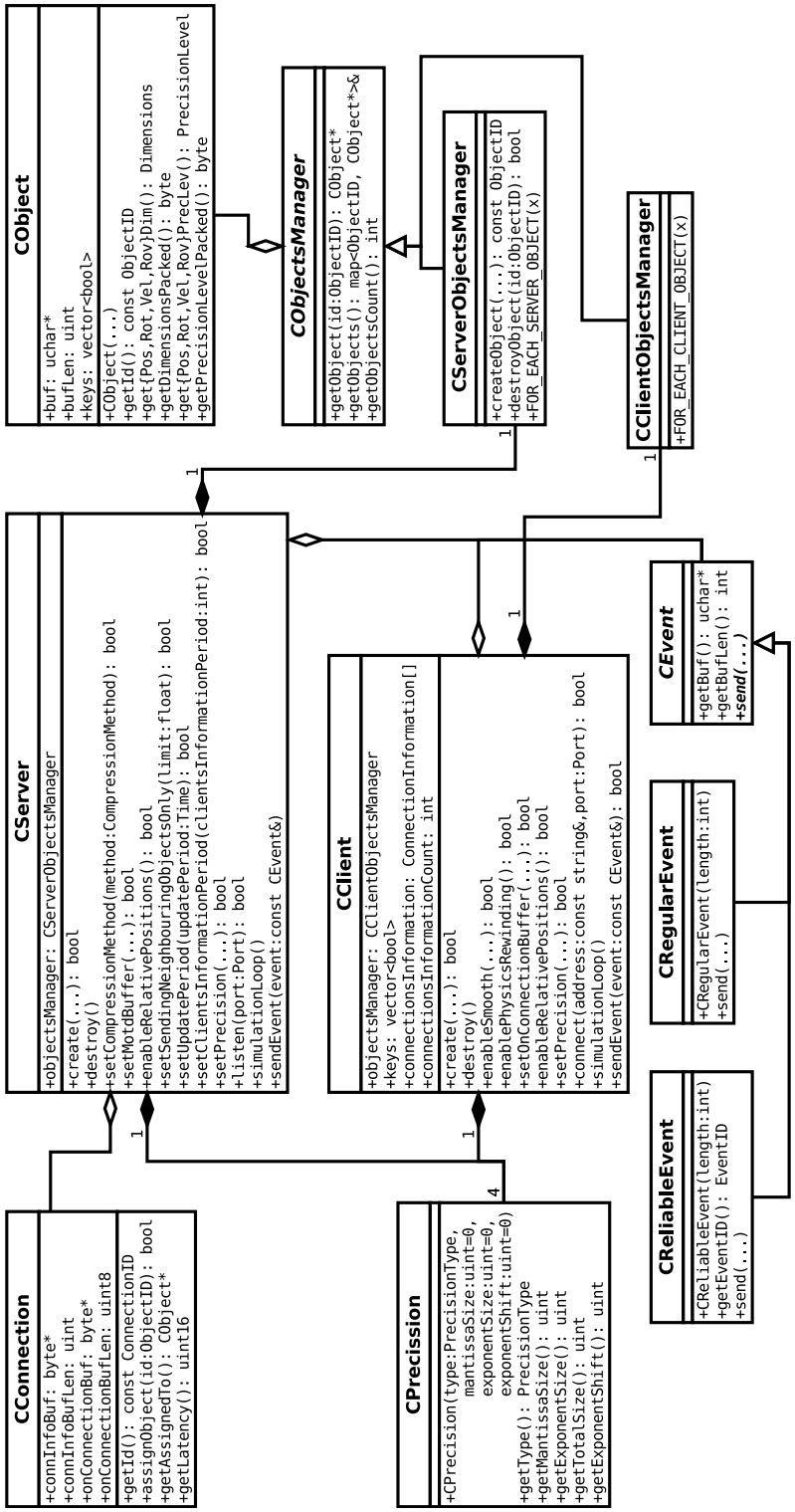


**Figure 4:** Class Diagram

```
bool CClient::enableSmooth(float posSmoothFactor, float posSmoothLimit, float rotSmoothFactor,
     float rotSmoothLimit, float velSmoothFactor, float velSmoothLimit, float rovSmoothFactor,
     float rovSmoothLimit);
bool CClient::enablePhysicsRewinding();
bool CClient::setOnConnectionBuffer(const byte* onConnectionBuf, uint onConnectionBufLen);
bool CClient::enableRelativePositions();
bool CClient::setPrecision(const CPrecision& first, const CPrecision& second, const
     CPrecision& third, const CPrecision& fourth);
```

When parameters are set up, the client calls `CClient::connect(const string&, Port)` function which connects to the server. If no error occurred, the client should proceed with the simulation — call `CClient::simulationLoop()` and `CClient::destroy()` at the end.

Every function, which returns `bool` type, returns `false` when an error has occurred. Otherwise, `true` is returned.

Code documentation, which is distributed with source code of the library, describes all the classes and methods in details.

## 4.3 Network Protocol

New network protocol has been designed for the project. It operates in the application layer of OSI model, although it provides features such as establishing connections or time-outs (which belong to lower layers).

Bazinga network protocol is built on top of UDP protocol. The first byte of each datagram denotes the type of the message that is being transmitted. Following subsections describe when which kind of message has to be sent. Detailed construction of every packet type is described in the section 4.3.6.

### 4.3.1 Establishing Connection Procedure

TCP protocol uses the so-called *Three Way Handshake* method in order to establish TCP session (connection) [9]. Programs which use Bazinga engine perform a similar procedure, yet the protocol involves also some higher level data transfer. Figure 5 intuitively presents which messages are being sent while establishing a new connection with the server.

SYN packet indicates an inclination of establishing connection. The server notes this fact and replies with SYNACK packet. Afterwards, the client sends CLIENT_INFO packet — containing information about the client. Server replies with SERVER_INFO (information about the server). Once the client receives SERVER_INFO, it prepares itself to participate in the simulation (e.g. initializes window, graphics). When the client is ready to start the simulation, it sends a LOADED packet, which is acknowledged with a WELCOME message — connection is established now.

If either side waits too long for the response, retransmission occurs. After several unanswered retransmissions, an error is returned.

If an error occurs at any stage of the connection procedure, the RST packet is being sent. Side which receives RST should immediately terminate the transmission and consider the connection as closed.

### 4.3.2 Data Exchange Phase

Once the connection is established, simulation data exchange starts. Every packet in this phase is being sent asynchronously with a specified frequency. The client sends KEYS packet, which comprises data about currently pressed keys on the keyboard. The server transmits UPDATE (description

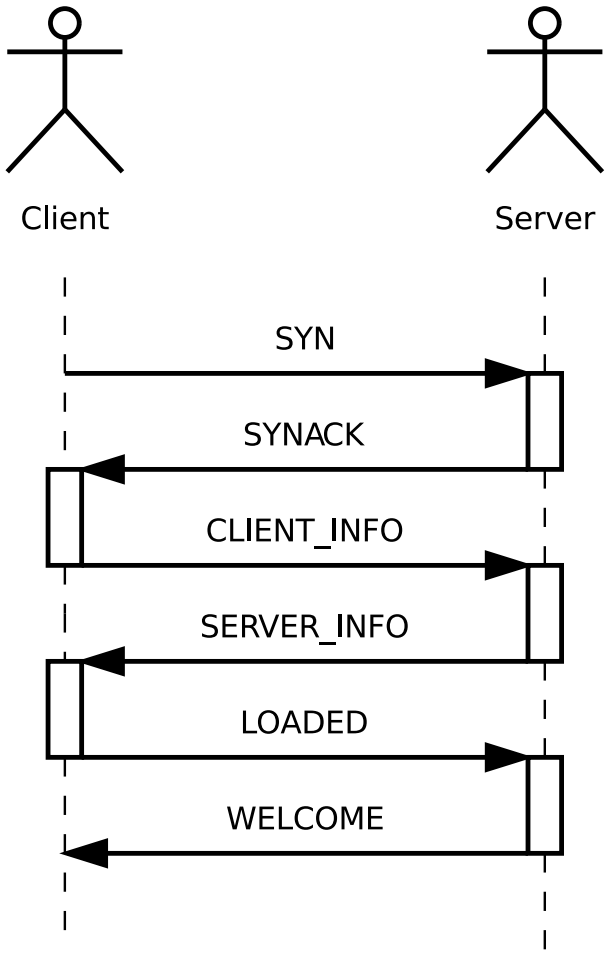


**Figure 5:** Establishing connection

of all objects that participate in the simulation) and CLIENTS_INFO — additional data about clients which is not so important and can be transmitted with less frequency.

### 4.3.3 Events

Two types of events are supported: regular and reliable. Some datagrams might be lost e.g. due to a network congestion. The side which sends a reliable event (RELIABLE_EVENT), waits for an acknowledgement (EVENT_ACK). After a definite period of time, when acknowledgment has not arrived, the event packet is deemed to be lost and subjected to retransmission. The reliable event is being retransmitted until the acknowledgment is received. Regular event (REGULAR_EVENT) is sent once and no acknowledgment is replied. Therefore, it is discouraged to put critical data within it.

Both server and client can send events.

### 4.3.4 Pings and Latency Measuring

The server sends PING messages occasionally in order to make sure that the client has not disconnected and measure the network latency. The client responds with PONG packet — the server notes the elapsed time interval.

With frequency defined by the application programmer, the server transmits additional data about clients with their network latencies (CLIENTS_INFO). Network games may use this packet to transmit nicknames and scores of players, as those values do not change very often.

### 4.3.5 Finalizing Connection

When a side wants to quit the simulation and terminate the connection, it sends FIN packet. The other side does not reply anyhow afterwards.

### 4.3.6 Packets Description

For the description of packets, BNF notation is used with the following data types (terminals):

- `byte` – one byte
- `int16` – two bytes in network byte order
- `int32` – four bytes in network byte order
- `0x00, ..., 0xff` – one byte written hexadecimally

Additionally, `buffer` type is convenient — one unsigned byte *n* denoting length of the buffer followed by *n* bytes of the buffer data. It is formally defined as follows:

```
buffer ::= <n> byte{n}
<n> ::= byte
```

As shown in the above example, the following convention is used: number in curly brackets denotes the number of repetitions of the symbol.

The root of the BNF tree is `NETWORK_PACKET`:

```
<NETWORK_PACKET> ::= <SYN> | <SYNACK> | <CLIENT_INFO> | <SERVER_INFO> | <LOADED> | <WELCOME> |
    <KEYS> | <UPDATE> | <CLIENTS_INFO> | <REGULAR_EVENT> | <RELIABLE_EVENT> | <EVENT_ACK> |
    <PING> | <PONG> | <FIN> | <RST>
```

The above rule simply lists types of packet types. Each of these packet types has the first byte reserved for the sake of explicitness.

```
<SYN> ::= <P_SYN> <client_id>
<P_SYN> ::= 0x00
<client_id> ::= int16
```

The client has to draw a random 16-bit identification number (`client_id`) before connecting. It is contained in `SYN` packet. This number will be used throughout the whole communication process to identify the client.

```
<SYNACK> ::= <P_SYNACK> <client_id>
<P_SYNACK> ::= 0x01
```

SYNACK packet comes from the server and indicates that the server agrees to accept the client under specified ID.

```
<CLIENT_INFO> ::= <P_CLIENT_INFO> <client_id> <keysCount> <relativePos>
    <precisionDescription>{4} <onConnectionBuffer>
<P_CLIENT_INFO> ::= 0x05
<keysCount> ::= byte
<relativePos> ::= byte
<precisionDescription> ::= <precType> <precMantissaSize> <precExponentSize> <precExponentShift>
<precType> ::= byte
<precMantissaSize> ::= byte
<precExponentSize> ::= byte
<precExponentShift> ::= byte
<onConnectionBuffer> ::= buffer
```

The client sends information about its properties with CLIENT_INFO message. It specifies the number of keys that, when pressed, are significant to the simulation, and therefore are being synchronized through the network (keysCount). relativePos is 0x01 if transfered positions of objects are to be expressed as relative to the observer; 0x00 otherwise.

Four precisionDescription blocks of data describe floating-point precision levels. Each of them consists of precType (0x00 for float, 0x01 for double, 0x02 for custom), precMantissaSize, precExponentSize, precExponentShift (three basic floating-point precision parameters which where described in section 3.5.2). If precType denotes float or double types, other parameters are insignificant and set to zero.

In addition, the packet contains onConnectionBuffer — general purpose, binary buffer which, by design, should contain an additional information about the client itself.

```
<SERVER_INFO> ::= <P_SERVER_INFO> <client_id> <compressionMethod> <motd> <updatePeriod>
<P_SERVER_INFO> ::= 0x06
<compressionMethod> ::= byte
<motd> ::= buffer
<updatePeriod> ::= int32
```

The server replies with SERVER_INFO, which comprises compressionMethod (0x00 for no compression, 0x01 for standard compression, 0x02 for delta-compression), motd buffer ("MOTD" stands for Message of the Day) and updatePeriod — period server sends UPDATE packets with (expressed in milliseconds).

```
<LOADED> ::= <P_LOADED> <client_id>
<P_LOADED> ::= 0x07
```

LOADED message states that the client is ready to participate in the simulation. It does not carry any more information besides the client's identification number.

```
<WELCOME> ::= <P_WELCOME> <client_id>
<P_WELCOME> ::= 0x08
```

The server acknowledges that the client entered the simulation with WELCOME message.

```
<KEYS> ::= <P_KEYS> <client_id> <keysSeq> <keysData>
<P_KEYS> ::= 0x0a
<keysSeq> ::= int32
<keysData> ::= byte{ceil(keysCount/8)}
```

The client constantly sends KEYS packets to inform the server which keys on the keyboard are being currently pressed. Packets are numbered with keysSeq field so that the server can drop packets which are received in the wrong order. keysData comprises keysCount bits which are packed into $\lceil \frac{\texttt{keysCount}}{8} \rceil$ bytes. Therefore, up to 7 oldest bits of the last byte may be unused and therefore must be set to zeros.

```
<UPDATE> ::= <P_UPDATE> <client_id> <updateSeq> <assignedTo> <objects>
<P_UPDATE> ::= 0x09
<updateSeq> ::= int32
<assignedTo> ::= int16
<objects> ::= <objectsCount> <object>{objectsCount}
<objectsCount> ::= byte
<object> ::= <object_id> <dimensions> <precisionLevels> <pos> <rot> <vel> <rov> <buf>
     <keysData>
<object_id> ::= int16
<dimensions> ::= byte
<precisionLevels> ::= byte
<buf> ::= buffer
```

`UPDATE` is the most substantial packet — it describes properties of all objects in the simulation. It is being sent constantly by the server with specified period. `updateSeq` numerates the packets so that packet received the in wrong order can be dropped by the client. `assignedTo` is an identification number of the object which the client is assigned to. Up to 255 objects can be synchronised. Each of object's description consists of two special bytes (`dimensions` and `precisionLevels`) followed by the properties of the object: position (`pos`), rotation (`rot`), velocity (`vel`), rotational velocity (`rov`), multi-purpose binary buffer (`buf`) and information about pressed keys (`keysData`, represented exactly as in `KEYS` packet).

Both `dimensions` and `precisionLevels` bytes are divided into 4 parts — 2 bits for each part. Figure 6 demonstrates this partition. The first part (less significant bits) is reserved for position, second for rotation, third for velocity and fourth (the most significant bits) for rotational velocity. Each pair of bits can take four different values: 00, 01, 10, 11. Pairs in `dimensions` byte denote respectively 0D, 1D, 2D, 3D — number of dimension of the property. Pairs in `precisionLevels` byte denote respectively the first, second, third and fourth precision level (all four precision levels have been agreed during the connection establishing).

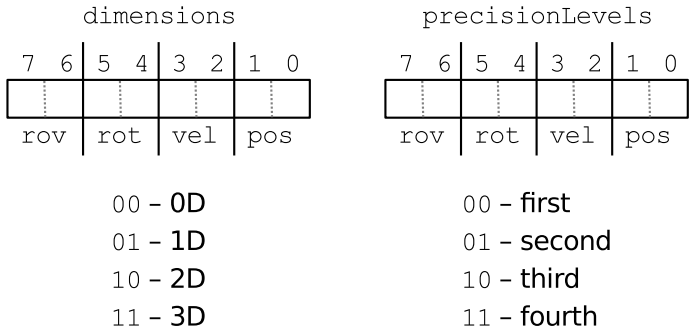


**Figure 6:** Denotation of individual bits in `dimensions` and `precisionLevels` bytes

Defining `pos`, `rot`, `vel` and `rov` strictly in BNF notation is quite problematic. The resulting rules would be certainly long, tedious and unreadable. Length and interpretation of each property is dependant on the value of proper pairs of bits in `dimensions` and `precisionLevels` bytes and the definition of the precision levels.

```
<CLIENTS_INFO> ::= <P_CLIENTS_INFO> <client_id> <clientsCount> <client>{clientsCount}
<P_CLIENTS_INFO> ::= 0x05
```

```
<clientsCount> ::= byte
<client> ::= <clientBuffer> <latency>
<clientBuffer> ::= buffer
<latency> ::= int16
```

CLIENTS_INFO packets are being sent much rarer than UPDATE messages and provide an additional information about all clients which participate in the simulation. A description of each client consists of a binary multi-purpose buffer named clientBuffer and 16-bit integer latency which estimates the network lag between the client and the server.

```
<REGULAR_EVENT> ::= <P_REGULAR_EVENT> <eventData>
<P_REGULAR_EVENT> ::= 0x0b
<eventData> ::= buffer


<RELIABLE_EVENT> ::= <P_RELIABLE_EVENT> <eventID> <eventData>
<P_RELIABLE_EVENT> ::= 0x0c
<eventID> ::= int16


<EVENT_ACK> ::= <P_EVENT_ACK> <eventID>
<P_EVENT_ACK> ::= 0x0d
```

Events mechanism has been precisely described in 4.3.3 section. Note that REGULAR_EVENTs do not need any identification numbers, whereas RELIABLE_EVENTs and the acknowledgments do.

```
<PING> ::= <P_PING> <client_id> <data>
<P_PING> ::= 0x03
<data> ::= int16


<PONG> ::= <P_PONG> <client_id> <data>
<P_PONG> ::= 0x03
```

The server uses PING packets to measure the time between the packet transmision and the reception time of PONG.

```
<FIN> ::= <P_FIN>
<P_FIN> ::= 0x02


<RST> ::= <P_RST>
<P_RST> ::= 0x04
```

FIN packet denotes a request of graceful connection termination. RST message indicates that a fatal error has occured and the side has dropped the connection.

# 5 Implementation

Bazinga engine consists of a total of 5623 lines of C++ code (4001 in source files and 1622 in headers). Efforts were made to write the library in a cross-platform way, so that it runs on as many operating systems as possible. It was tested under Windows and Linux only, although it should compile and run with no major problems on other UNIX-like operating systems. No integrated development environment was used, so the library should compile without any third-party software besides basic GNU tools. Makefile is supplied with the source code to make the compilation process easy.

## 5.1 Examples

An example application is developed along with the library source code. It demonstrates the usage of the network engine — it is recommended to study the example carefully before developing own network simulation using Bazinga. The example application comprises two one-file programs: `server_example.cpp` and `client_example.cpp`.

In order to test the simulation via a computer network, `server_example` program must run on one of the machines. After it is started, client machines may run `client_example` program (server's IP address must be passed through the command line argument).

Sample programs provide very simple physics simulation that involves multiple balls, which move without gravity and collide with each other. When a client connects, a new coloured ball is created. The client can apply force to his ball — it colours all the balls it collides with.

Two big listings will be discussed in this section. These are fragments of the example programs (a bit shortened to emphasize how to use the library's API and not to focus on technical details). Full, compilable source files are disturbed with the source code of Bazinga library in the `example` directory.

### 5.1.1 The Server Example

```
1  bool physics(Time last, Time rightNow)
2  {
3      float d = float(rightNow - last)/10000;
4
5      FOR_EACH_SERVER_OBJECT (x)
6          /* [...] physics goes here */
7      FOR_EACH_SERVER_OBJECT_END
8
9      // collisions
10     FOR_EACH_SERVER_OBJECT (x)
11         FOR_EACH_SERVER_OBJECT (y)
12             /* [...] collisions go here */
13         FOR_EACH_SERVER_OBJECT_END
14     FOR_EACH_SERVER_OBJECT_END
15
16
17     // broadcast events randomly
18     if (rand()%500 == 0)
19     {
20         CRegularEvent ev(11);
21         //CReliableEvent ev(11);
22         strncpy((char*)ev.getBuf(), "HelloWorld", 11);
23
24         Server.sendEvent(ev);
25     }
26
27     return true;
28 }
29
30
31
32 bool clientConnected(CConnection* c)
33 {
34     // create new object for the client
35     CObject* obj;
36     ObjectID clientObject = Server.objectsManager.createObject(
37         Dimensions::Dim2D, Dimensions::Dim0D, Dimensions::Dim2D, Dimensions::Dim0D,
38         PrecisionLevel::Second, PrecisionLevel::First, PrecisionLevel::Third,
              PrecisionLevel::First);
39
40     cout << "Created␣object␣is␣" << uint(clientObject) << endl;
```

```
41
42      if ((obj = Server.objectsManager.getObject(clientObject)) == NULL)
43          return false;
44
45      // assign positions and buffers
46      obj->pos.x = float(rand()%600 + 100);
47      obj->pos.y = float(rand()%600 + 100);
48      obj->vel.x = float(rand()%100)/100;
49      obj->vel.y = float(rand()%100)/100;
50
51      static int index = 1;
52      obj->buf = new byte[2];
53      obj->buf[0] = index++;
54      obj->buf[1] = 1;
55      obj->bufLen = 2;
56
57      if (index == 7)
58          index = 1;
59
60      // assign object to the client
61      c->assignObject(clientObject);
62
63      // copy 'on connection buffer' to 'connection information' buffer (nickname)
64      c->connInfoBufLen = c->onConnectionBufLen;
65      c->connInfoBuf = new byte[c->connInfoBufLen];
66      memcpy(c->connInfoBuf, c->onConnectionBuf, c->onConnectionBufLen);
67      return true;
68  }
69
70
71
72  bool clientDisconnected(CConnection* c)
73  {
74      CObject* assignedTo = c->getAssignedTo();
75      if (assignedTo)
76          Server.objectsManager.destroyObject(assignedTo->getId());
77      return true;
78  }
79
80
81
82  void eventsProcessor(const CEvent& e, ConnectionID id)
83  {
84  }
85
86
87
88  int main()
89  {
90      // ********************************************************************
91      // * Begin of Server Configuration
92      // ********************************************************************
93
94      if (!Server.setCompressionMethod(CompressionMethod::DeltaCompression))
95          exit(1);
96
97      Server.setPrecision(
98          CPrecision(PrecisionType::IEEE_float),
99          CPrecision(PrecisionType::Custom, 11, 4, 0),
100         CPrecision(PrecisionType::Custom, 11, 4, 7),
101         CPrecision(PrecisionType::IEEE_float)
102         );
103
104     if (!Server.create(5, eventsProcessor, physics, clientConnected, clientDisconnected))
105         exit(1);
106
107     // ********************************************************************
108     // * End of Server Configuration
109     // ********************************************************************
```

```
110
111     // create 25 balls at the beginning
112     for (int i=0; i<25; i++)
113     {
114         CObject* obj;
115         ObjectID clientObject = Server.objectsManager.createObject(
116             Dimensions::Dim2D, Dimensions::Dim0D, Dimensions::Dim2D, Dimensions::Dim0D,
117             PrecisionLevel::Second, PrecisionLevel::First, PrecisionLevel::Third,
                    PrecisionLevel::First);
118
119         cout << "Created␣object␣is␣" << uint(clientObject) << endl;
120
121         if ((obj = Server.objectsManager.getObject(clientObject)) == NULL)
122             return false;
123
124         obj->pos.x = float(rand()%600 + 100);
125         obj->pos.y = float(rand()%600 + 100);
126         obj->vel.x = float(rand()%100)/100 - 0.5f;
127         obj->vel.y = float(rand()%100)/100 - 0.5f;
128
129         obj->buf = new byte[2];
130         obj->buf[0] = 0;
131         obj->buf[1] = 0;
132         obj->bufLen = 2;
133     }
134
135     if (!Server.listen(41516))
136     {
137         Server.destroy();
138         return 1;
139     }
140
141     Server.simulationLoop();
142
143     Server.destroy();
144
145     return 0;
146 }
```

**Listing 1:** The most important parts of `server_example.cpp` file

`physics` (lines 1–28 of Listing 1) is the function where the whole physics simulation is implemented in. Movement of each object has been removed from the listing (line 6) as it is long and not connected directly with the topic. This sample uses the Euler method to solve trivial motion equations. It was chosen for simplicity, yet in most cases more advanced methods need to be used to approximate differential equations solutions (like RK4 method described in [2]). Collision detection and handling is also cut (it is based on the equations and source code from [10]).

`physics` function also deals with sending events. With low probability, the server creates a new `CRegularEvent` object, fills it with data and transmits to every connected client. `CReliableEvent` can be used instead — the server would demand the acknowledgment that the message was not lost then. The event object does not have to be created dynamically.

When a new client establishes a connection, the `clientConnected` function is called. It creates a new object, assigns it to the client and sets the object's default parameters. It allocates multi-purpose buffer dynamically (line 52). It is assumed that the client puts his nickname in `onConnectionBuf`. In lines 640–66 memory for `connInfoBuffer` is reserved and nickname is copied there so that other clients will receive information about the client's nickname.

Lines 720–78 provide the procedure that is called when the client disconnects. It simply destroys the client's object.

The server does not have to receive any events, so `eventsProcessor` routine is empty.

`main` function is of course a place where the program starts its execution. First, compression

method is set (this sample uses delta-compression). Then, precision levels are used. Only three first levels are actually used, so the fourth one is insignificant. Finally, the server is created with `CServer::create` method. Note that function pointers are passed as arguments.

The server also creates 25 new balls before it starts. It sets all the default values, similarly to as it is done when a new client connects.

When everything is set up, the server can listen for new connections. The port number (41516) is passed as an argument to the `CServer::listen` function. Immediately afterwards, the main loop of the simulation is called. Most of the application time is spent in this function. When the simulation is finished (that is when `physics` returns `false`), the function returns and the server is destroyed and the program exits with error code `0`.

### 5.1.2 The Client Example

```
1  bool physics(Time last, Time rightNow)
2  {
3      float d = float(rightNow - last)/10000;
4
5      FOR_EACH_CLIENT_OBJECT (x)
6          /* [...] physics goes here */
7      FOR_EACH_CLIENT_OBJECT_END
8
9      // collisions
10     FOR_EACH_CLIENT_OBJECT (x)
11         FOR_EACH_CLIENT_OBJECT (y)
12             /* [...] collisions go here */
13         FOR_EACH_CLIENT_OBJECT_END
14     FOR_EACH_CLIENT_OBJECT_END
15
16
17     // print information to the stdout
18     static Time displayInfoCounter = 0;
19     displayInfoCounter += rightNow - last;
20     if (displayInfoCounter >= 1000000)
21     {
22         displayInfoCounter -= 1000000;
23
24         // clear console;
25         for (int i=0; i<25; i++)
26             cout << endl;
27
28         cout << "List␣of␣players␣and␣latencies:" << endl;
29
30         for (int i=0; i<Client.connectionsInformationCount; i++)
31         {
32             for (uint j=0; j<Client.connectionsInformation[i].bufLen; j++)
33                 cout << Client.connectionsInformation[i].buf[j];
34             cout << ":␣" << Client.connectionsInformation[i].latency << endl;
35         }
36     }
37
38     return true;
39 }
40
41
42
43 bool render()
44 {
45     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
46
47     glMatrixMode(GL_MODELVIEW);
48     glLoadIdentity();
49
50     FOR_EACH_CLIENT_OBJECT (x)
```

```
51          /* [...] draw ball at x->pos */
52      FOR_EACH_CLIENT_OBJECT_END
53
54      SDL_GL_SwapBuffers();
55
56      return true;
57  }
58
59
60
61  bool input()
62  {
63      SDL_Event event;
64
65      while (SDL_PollEvent(&event))
66      {
67          switch (event.type)
68          {
69          case SDL_KEYDOWN:
70              if (event.key.keysym.sym == SDLK_w)
71                  Client.keys[0] = true;
72              if (event.key.keysym.sym == SDLK_s)
73                  Client.keys[1] = true;
74              if (event.key.keysym.sym == SDLK_a)
75                  Client.keys[2] = true;
76              if (event.key.keysym.sym == SDLK_d)
77                  Client.keys[3] = true;
78              if (event.key.keysym.sym == SDLK_SPACE)
79                  Client.keys[4] = true;
80
81              if (event.key.keysym.sym == SDLK_ESCAPE)
82                  return false;
83              break;
84
85          case SDL_KEYUP:
86              if (event.key.keysym.sym == SDLK_w)
87                  Client.keys[0] = false;
88              if (event.key.keysym.sym == SDLK_s)
89                  Client.keys[1] = false;
90              if (event.key.keysym.sym == SDLK_a)
91                  Client.keys[2] = false;
92              if (event.key.keysym.sym == SDLK_d)
93                  Client.keys[3] = false;
94              if (event.key.keysym.sym == SDLK_SPACE)
95                  Client.keys[4] = false;
96              break;
97
98          case SDL_QUIT:
99              return false;
100         }
101     }
102     return true;
103 }
104
105
106
107 bool load()
108 {
109     return true;
110 }
111
112
113
114 bool prepareGraphics()
115 {
116     /* [...] initalize OpenGL and window in SDL, return false on error */
117     return true;
118 }
119
```

```
120
121
122 void eventsProcessor(const CEvent& e)
123 {
124     cout << "Got␣event,␣message␣is:␣" << e.getBuf() << endl;
125 }
126
127
128
129 int main(int argc, char* argv[])
130 {
131     // ********************************************************************
132     // * Begin of Client Configuration
133     // ********************************************************************
134
135     if (!Client.enableSmooth(0.9f, 200.0f, 1.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f))
136         exit(1);
137
138     if (!Client.enablePhysicsRewinding())
139         exit(1);
140
141     if (!Client.setOnConnectionBuffer(
142         argc <= 2 ? (const byte*)"unknown" : (const byte*)argv[2],
143         argc <= 2 ? 7 : strlen(argv[2])))
144         exit(1);
145
146     Client.setPrecision(
147         CPrecision(PrecisionType::IEEE_float),
148         CPrecision(PrecisionType::Custom, 11, 4, 0),
149         CPrecision(PrecisionType::Custom, 11, 4, 7),
150         CPrecision(PrecisionType::IEEE_float)
151         );
152
153     if (!Client.create(5, eventsProcessor, physics, render, input, load))
154         exit(1);
155
156     // ********************************************************************
157     // * End of Client Configuration
158     // ********************************************************************
159
160     if (!Client.connect(argc > 1 ? argv[1] : "localhost", 41516))
161     {
162         Client.destroy();
163         return 1;
164     }
165
166     if (!prepareGraphics())
167     {
168         Client.destroy();
169         return 1;
170     }
171
172     Client.simulationLoop();
173
174     Client.destroy();
175     SDL_Quit();
176
177     return 0;
178 }
```

**Listing 2:** The most important parts of `client_example.cpp` file

`physics` routine (lines 1–39 of Listing 2) on the client's side is very similar. However, no events are being sent. Additionally, information about all the clients is being displayed once every second.

`render` function (lines 43–57) is being called repeatedly to draw graphics on the computer screen. It uses OpenGL calls to draw each object. The drawing code has been removed from the listing, for it is not closely related to the network issues.

Keyboard events are handled in `input` function with SDL mechanisms. Vector `Client.keys` is being updated there so that Bazinga engine can inform the server about keys which are pressed on the client's keyboard.

The sample program is so simple that it does not require any files to be loaded on connection. In most cases, some files need to be loaded (such as 3D models, textures). It should be done inside the `load` function.

The content of `prepareGraphics` is not so interesting, so it has been cut.

The client handles coming events in `eventsProcessor` function. It merely prints the message content to the standard output.

`main` function, like on the server side, prepares the communication parameters and enables important mechanisms. First, smoothing algorithm is enabled (and proper constants are defined). Then, physics rewinding algorithm is enabled. Later, connection buffer is set (it contains nickname, which is passed through a command-line argument. If no argument was passed, "`unnamed`" value is presumed). Precision levels must match with those defined by the server. After such preparations, the client can initialize `Client` object and connect to the server.

The client stays in the `simulationLoop` until `input`, `physics` or `render` function return `false` or a problem on the server's side occurs.

# 6 Discussion and Conclusions

The goal of this paper has been achieved. Bazinga's design is robust and the implementation works satisfactorily. Developing real-time network simulations and games with this library is much easier. A fair compromise has been reached between functionality and generality — the programmer can easily adjust multi-functional buffers to transmit any kind of data, according to the current need. Easy to understand examples and basic code documentation have been carried out along with the library, which makes its usage much more comfortable.

Nevertheless, certain parts of the library have not been fully worked out yet. Some effort still has to be made in order to improve:

- `double` type handling (*Blossom Math* does not support `vec3` class based on the double precision floating-point numbers yet)
- subnormal numbers (numbers which are very close to the zero are packed as zero in current implementation)
- fragmentation — if there are too many objects, it may happen that the `UPDATE` packet is too large to be transmitted through the network. It should be fragmented into several smaller parts then and assembled at the reception. The current project simply assumes that this problem does not occur
- the client's awareness about the possible loss of connection. In the current implementation, the client may not notice the problem when the server suddenly stops responding (e.g. due to a major network problem)

Practical experiments and tests have been conducted. Unfortunately, it is difficult to clearly gather the results in a table or plot, because achieved smoothness highly depends on the properties of a specific application. Some physics simulations may be more chaotic than others (physics

routine is chaotic when small differences in initial conditions yield much larger divergences in results). The more chaotic a simulation is, the more vulnerable to network latencies is the application. However, assuming completely non-chaotic movement, physics rewinding algorithm provides an excellent smoothness despite even very irregular and high network latencies (whereas naive method fails).

Studying a behaviour of the sample application has shown that the naive method (when neither physics rewinding nor smoothing algorithm is enabled) is unsatisfactory at average latencies of 100 milliseconds. With aforementioned mechanisms enabled, a similar level of smoothness-loss is noticeable when latencies reach 350 milliseconds.

# References

[1] SDL Documentation. `http://libsdl.org/`.

[2] Glenn Fiedler. Integration Basics, September 2006. `http://gafferongames.com/game-physics/integration-basics/`.

[3] Glenn Fiedler. Networked Physics, September 2006. `http://gafferongames.com/game-physics/networked-physics/`.

[4] Brian "Beej Jorgensen" Hall. Beej's Guide to Network Programming. 2009.

[5] IEEE Std. 754™-2008 Standard for Floating-Point Arithmtic, August 2008.

[6] Basic Reference Model for Open System Interconnection. Int. Standards Org., 1984.

[7] Jean-loup Gailly, Mark Adler. zlib. `http://www.zlib.net/`.

[8] P.V. Mockapetris. Domain names - implementation and specification. Number 1035 in Request for Comments. IETF, November 1987. `http://www.ietf.org/rfc/rfc1035.txt`.

[9] J. Postel. Transmission Control Protocol. Number 793 in Request for Comments. IETF, September 1981. `http://www.ietf.org/rfc/rfc793.txt`.

[10] Thomas Smid. Elastic and Inelastic Collision in Two Dimensions. `http://www.plasmaphysics.org.uk/collision2d.htm`.

[11] Wojciech Sterna. Blossom Math. `http://maxest.gct-game.net/blossom-engine/`.

[12] Bjarne Stroustrup. The C++ Programming Language, Second Edition. Addison-Wesley, 1991.

[13] M.T. Tham. Dealing with Measurement Noise, a Gentle Introduction to Noise Filtering. 1996-2009. `http://lorien.ncl.ac.uk/ming/filter/filewma.htm`.