Game Protocol

Christofer Nolander cnol@kth.se

Protocol

This document specifies the network protocol used for communication between the server and client. The protocol is inspired by a series of articles made by Glenn Fiedler (Gaffer on Games).

Unless otherwise specified, all integers in this document are assumed to be in network (big endian) byte order.

Transport

UDP is used as the transport layer of choice, this is due to its lower latency and lack of reliability and congestion control compared to TCP. It might at first seem counterintuitive that we do not want reliability, but consider that most things that happen in (real time) games are very time sensitive: knowing where a monster was 2 seconds ago is of no use to us than where it is now. TCP implements its reliability by resending packets until they are acknowledged by the receiver. This inherently introduces delays in the data stream when new packets are blocked on previous packets that failed to be acknowledged. In order to improve responsiveness in this protocol we instead use UDP and utilize its fire and forgive message passing.

However, sometimes we *do* want reliability, especially when first initializing the game state on each side or when sending important events (such as when a player lost or won): dropping a packet would result in a corrupted state or failed transaction. So what we really need is a way to toggle reliability on and off on a message by message basis.

Implementation

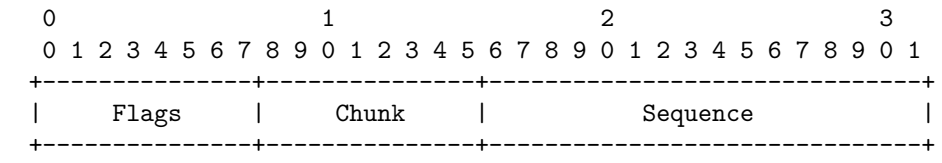
We can implement reliability by adopting a technique similar to TCP: resending unacknowledged packets. We do, however, need to make sure that unreliable messages are still sent even if we are stuck resending the same reliable packet over

and over. This is accomplished by multiplexing the data stream over multiple virtual channels called *sequences*.

Each sequence consists of up to 256 individual chunks, each chunk representing a small part of the total payload. When sending a message, the sender splits the payload into a number of these chunks and sends every single one to the receiver alongside the sequence id and the index of that chunk. When packets are received, the receiver can look at the sequence id and chunk index to reconstruct the complete payload.

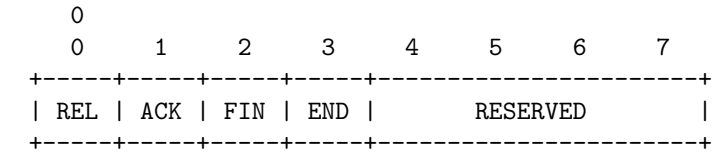
Since packets may arrive out of order we get multiplexing for free: the moment all chunks have arrived in a given sequence the payload can be reconstructed and used in the program.

Packet Header The packet header contains information for reconstructing the packets on the receiver. It is a total of 4 bytes:



- `Flags`: a bitfield containing various flags, see below.
- `Chunk`: the index of the chunk the packet describes.
- `Sequence`: the id of the sequence the packet belongs to.

Layout of `Flags`:



- `REL`: if set the packet is reliable and needs to be acknowledged.
- `ACK`: this packet acknowledges a previously sent packet.
- `FIN`: this packet contains the final chunk in its sequence.
- `END`: if set, the connection has closed.

Sending Packets A sequence consists of up to 256 chunks. Each chunk is a maximum of 504 bytes. This limit comes from the fact that the MTU is guaranteed to be at least 576 minimum. The largest IP header is 60 bytes, and the UDP header an additional 8 bytes. That leaves us with 508 bytes for the entire packet, and with our custom header, that takes up an additional 4 bytes we are left with 504 bytes for the chunk.

When splitting a payload into chunks, every chunk must be 504 bytes except for the last chunk which may be smaller. When sending packets to the receiver the sender must mark the packet that contains the last chunk in the sequence with the `FIN` flag.

Receiving Packets When a packet with the `FIN` bit set the receiver knows how many chunks are in the sequence (chunk indices 0 through `Chunk` inclusive). Once all chunks in a sequence have been received the payload may be handed off to the rest of the program. Care must be taken to avoid payload duplication leading to duplicate payloads being received.

Acknowledging Packets If a packet with the `REL` flag set is received the receiver should immediately send back another packet with the `ACK` flag set as well as `Chunk` and `Sequence` set to the same values as the original packet. This informs the sender that the packet has been received and shouldn't be retransmitted. However, due to the unreliability of UDP previously sent packets may still be received by the receiver.

When a packet with the `ACK` flag set is received the receiver must stop resending the packet with the same `Sequence` id and `Chunk` index as the incoming packet.

Connections

UDP is a connectionless protocol. In this game we want connections in order to manage clients easier. How we initialize a connection varies depending on if we are the server of client, but the process is straight forward:

> Note: for all the steps, see details in the sections below.

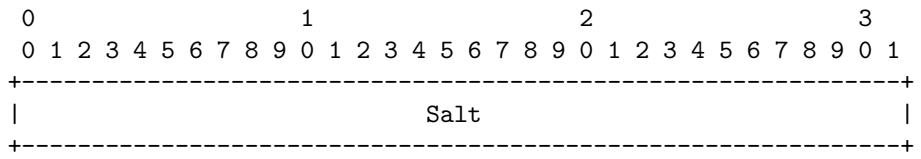
1. The client sends an `Init` message.
2. The server receives the `Init` message and responds with a `Challenge`.
3. The client receives the `Challenge` and responds with a `ChallengeResponse`.
4. The server receives the `ChallengeResponse` and verifies it.
5. If the verification succeeded, the connection is now open.

In order to close a connection either the client or server may send a packet with the `END` bit set.

If the client or server does not receive a packet from the other side for more than 15 seconds, the connection is considered closed.

Init

The `Init` message consists of a single 32-bit integer:

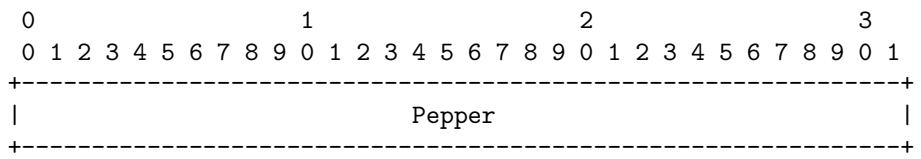


This integer is called the `Salt` and should be randomly chosen by the client for reasons that well be made clear below.

Challenge

The `Challenge` messages is used to challenge the authenticity of the connecting client. The IP and UDP header contains the source address of the packet, but before we start communicating we need to verify that the client actually connected from this address. So we challenge the client's claim by sending it a `Challenge`.

The `Challenge` consists of a single 32-bit integer:

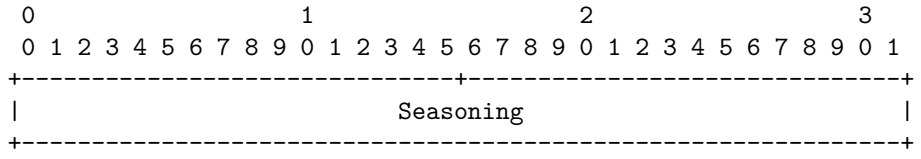


This integer is called the `Pepper` and is randomly chosen by the server.

ChallengeResponse

In order to make sure that the client that sent the `Init` request was the same as the one specified in the IP/UDP headers we require that the client has access to both the `Salt` and `Pepper` by asking them to XOR them together. The result can only be known if the client both sent the `Salt` from the specified address and can receive messages on that address as well.

The `ChallengeResponse` message consists of a single 32-bit integer:



This integer is called the `Seasoning` and is the XOR of the `Salt` and `Pepper`.

Verification

When the server receives the `ChallengeResponse` response from the client it can verify that the `Seasoning` is in fact the expected value (`Salt XOR Pepper`).

Binary Encoding of Messages

With a connection established we can finally start sending game related messages... almost. We still need a way to encode and decode the messages, both quickly and efficiently. To do this I developed a custom binary protocol I am calling Rabbit (derived from raw-bits). In order to keep message sizes down it is not self describing and uses both bit-packing and variable size integers.

Bitpacking

The concept "bitpacking" is rather simple, instead of seeing a payload as a list of bytes we seem them as a list of bits.

For example, if we wanted to encode a number we know to be always less than 32, say for example 14, we know we don't need any more than 5 bits to do so: `01110`. Compared to just encoding the number as a full byte, we saved ourselves a whopping 3 bits!!! That may not sound like much, but that is just 62.5% the original size, taken over a complete payload the percentages add up.

However in this encoding not only does byte order matter, but bit order as well. In Rabbit bits are pushed to the stream starting with the least significant bit (LSB) first. When the bitstream is sent over the wire the bitsream is padded with zero or more zeroes until its length is a multiple of 8. These bits are then grouped into continuous groups of 8 bits (bytes). These bytes are then sent across in order, starting with the byte that contains the LSB, ie. in little endian order.

Decoding the encoded data is just running this process in reverse.

Example Consider that we want to encode three numbers in order using Rabbit. We begin by converting them into binary:

```
15 using 5 bits ->   01111
81 using 7 bits -> 1010001
 1 using 2 bits ->      01
```

We then push the bits (LSB first) to the stream, starting from left to right:

```
Start:   ""
Push 15: "11110"
Push 81: "111101000101"
```

```
Push 1:  "11110100010110"
```

Then we pad the bits to a multiple of 8:

```
Start:   "11110100010110"
Pad:     "1111010001011000"
```

Then we can split the bits into bytes:

```
Start:   "1111010001011000"
Group:   "11110100" "01011000"
```

Finally we can send the bytes, in the order we get when reading left to right.

Variable length integers

Usually, numbers tend to be small. We regularly use 32-bit or even 64-bit integers, when not strictly necessary, only to get some extra redundancy. However, that wastes a lot of space that we want to reclaim. In order to do so we encode integers using a zig-zag variable length encoding.

Encoding Unsigned Integers When encoding a `n`-byte integer `i` we first determine the smallest number of bytes necessary to encode it if the highest bytes are implicitly padded with zeroes, provided that the case `i = 0` requires 1 byte. We call this number `m`. If we always implicitly require that an integer needs 1 byte to be encoded, we will need an additional `m - 1` additional bytes to encode the number `i`. Then we determine the smallest number of bits required to encode the number `n - 1`, call this number `k`. Since `m <= n` we can also encode `m - 1` using `k` bits. So we write the number `m - 1` to the bitstream using `k` bits. Following this we write the lowest `m` bytes of `i` to the stream in little endian order. The number `i` has now been encoded.

Decoding Unsigned Integers When decoding a `n`-byte integer `i` we first determine the smallest number of bits required to encode the number `n - 1`, call this number `k`. We then read the `k`-bit number `m` from the stream. Then we read `m + 1` bytes from the stream, which is our number `i`.

Encoding and Decoding Signed Integers Signed integers are encoded using twos complement and a zig-zag pattern. The reason we special case signed integers is that negative numbers, when encoded as unsigned integers, are really large: `-1` as a 8-bit number is 255 (or `11111111`). Since numbers closer to 0 are more common we use a trick to interleave positive and negative integers:

```
Encoded      Decoded
-------      -------
 0            0
```

```
1           -1
2            1
3           -2
4            2
5           -3
...
4294967294   2147483647
4294967295  -2147483648
```

As can be seen in the table above, we can simply encode negative numbers as unsigned integers compactly in this variable length encoding if we interleave them first.

For details on how to convert between them, see: https://developers.google.com/protocol-buffers/docs/encoding?csw=1#signed-integers

Encoding/Decoding Floating Point Numbers 32 and 64 bit floating point numbers are encoded and decoded as 32 bits with the LSB of the fractional part first according to the IEEE 754 standard.

Messages

Messages are sent between the client and server in order to exchange information. The messages consists of various data structures encoded using the Rabbit protocol.

The client may send `Request`s to the server using a specific channel id. The channel will then respond with a `Response` using that same channel id.

Conventions

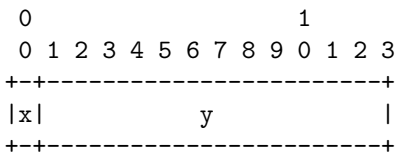
In order to define the encoding of messages, and keep this section readable, the components of every message is going to be defined in a list. Components of the messages are encoded in the same order as they are listed and their types.

Types are either named messages within code blocks, `Type`, or a number prefixed by either 'u' for unsigned integers, 'i' for signed integers or 'f' for floating point numbers.

For example:

- `x` (u1)
- `y` (i12)

would encode a message that consists of two integers that are a 1-bit unsigned integer and a 12-bit signed integer, respectively:



Components may also be conditionally included like so:

- `x` (u1)
- `y` (if `x` = 0 then u8)
- `y` (if `x` = 1 then f32)
- `z` (if `x` = 1 then `Data`)

The above tells us that the if `x` is not set then the message only has the 8-bit integer `y`. If `x` is set then `y` is a 32-bit float which is followed by `z`, a nested message called `Data`.

Additionally repetitions may be specified like so:

- `count` (u32): number of numbers in the list.
- `list` (`count` * u32): a list of `count` 32-bit unsigned integers.

ServerMessage

A message coming from the server: either an `Event` or `Response`.

Encoding:

- `variant` (1-bit integer): the kind of message
- `body` (if `variant` = 0 then `Event`)
- `body` (if `variant` = 1 then `Response`)

Event

An event occured at a specific time.

Encoding:

- `time` (u32): the tick index at which the event happened
- `kind` (`EventKind`): the kind of event that occured

EventKind

A specific event that happened.

Encoding:

- `variant` (u1)
- `body` (if `variant` = 0 then `Snapshot`): a snapshot of the current game state
- `body` (if `variant` = 1 then `GameOver`): the game was won/lost

GameOver

The result of a game. Tells the client if it won the game or lost.

Encoding:

- `variant` (u1): if 0, the client lost. If 1, the client won.

Snapshot

A snapshot of the current game state. Contains the state of each entity in the world.

Encoding:

- `count` (u32): the number of entities in the world.
- `entities` (`count` * `Entity`): All entities in the world.

Entity

A single entity in the world.

Encoding:

- `id` (u32): the id of the entity.
- `kind` (`EntityKind`): the kind of entity.

EntityKind

Different kinds of entities.

Encoding:

- `variant` (u2)
- `body` (if `variant` = 0 then `Object`)
- `body` (if `variant` = 1 then `Player`)
- `body` (if `variant` = 2 then `Dead`)

Object

An object in the world.

Encoding:

- `position` (`Point`): the location of the object in the world
- `kind` (u1): if 0 then the object is a tree, otherwise it is a mushroom
- `breakable` (u1): 1 if the entity can be broken and picked up
- `durability` (if `breakable` = 1 then f32): how much durability is left until the object can be picked up by a player.
- `health` (u32): the current health of an object
- `max_health` (u32): the maximum amount of health of an object

Player

A player in the world.

Encoding:

- `position` (`Point`): the location of the object in the world
- `movement` (`Direction`): the direction the player is moving
- `is_holding` (u1)
- `holding` (if `is_holding` = 1 then u32): the id of the entity currently held by the player
- `is_breaking` (u1)
- `breaking` (if `is_breaking` = 1 then u32): the entity currently being broken by the player
- `owner` (u32): the id of the player controlling this specific player
- `health` (u32): the current health of an object
- `max_health` (u32): the maximum amount of health of an object

Dead

The entity has died or was destroyed.

Encoding:

Empty

Direction

The direction in the world, as seen from above, with north aligned with the positive y-axis.

Encoding:

- `bits` (u8): a bitfield specifying the direction:
  - if bit 0 is set, the direction points north.
  - if bit 1 is set, the direction points west.
  - if bit 2 is set, the direction points south.
  - if bit 3 is set, the direction points east.

Point

An position in the world. The x-axis is given from left to right, the y-axis backward to forwards and the z-axis from below upwards.

Encoding:

- `x` (f32): the x-coordinate in the world
- `y` (f32): the y-coordinate in the world
- `z` (f32): the z-coordinate in the world

Response

A response to a `Request` made by a client.

Encoding:

- `channel` (u32): the id of the request this is a response to.
- `kind` (`ResponseKind`)

ResponseKind

The type of response.

Encoding:

- `variant` (u2)
- `body` (if `variant` = 0 then `Error`)
- `body` (if `variant` = 1 then `Pong`)
- `body` (if `variant` = 2 then `Connect`)

Error

The request failed.

Encoding:

- `length` (u32): the length of the error message
- `text` (`length` * u8): a UTF-8 encoded error message.

Pong

A response to a ping, may be used to calculate latency or to stop the connection from timing out.

Encoding:

Empty

Connect

The player connected to the game session.

Encoding:

- `player` (u32): the player id assigned to this client.
- `snapshot` (`Snapshot`): the current state of the game.

ClientMessage

A message sent from the client to the server.

Encoding:

- `variant` (u32)
- `body` (if `variant` = 0 then `Request`)
- `body` (if `variant` = 1 then `Action`)

Request

A request from the client to the server.

Encoding:

- `channel` (u32): the channel to receive the response from.
- `kind` (`RequestKind`): the kind of request

RequestKind

Specifies a certain kind of request.

Encoding:

- `variant` (u1)
- `body` (if `variant` = 0 then `Ping`): request a `Pong` from the server.
- `body` (if `variant` = 1 then `Init`): request to join the game session.

Action

The client performed an action.

Encoding:

- `variant` (u2)
- `body` (if `variant` = 0 then `Break`)
- `body` (if `variant` = 1 then `Throw`)
- `body` (if `variant` = 2 then `Move`)

Break

The client requested to break an entity.

Encoding:

- `is_breaking` (u1): should an entity be broken or not?
- `entity` (if `is_breaking` = 1 then u32): the entity to break.

Throw

The client requested to throw the currently held entity.

Encoding:

- `target` (`Point`): the location to throw the entity towards.

Move

The client requested to move in a certain direction.

Encoding:

- `direction` (`Direction`): the direction to move in.

The Client

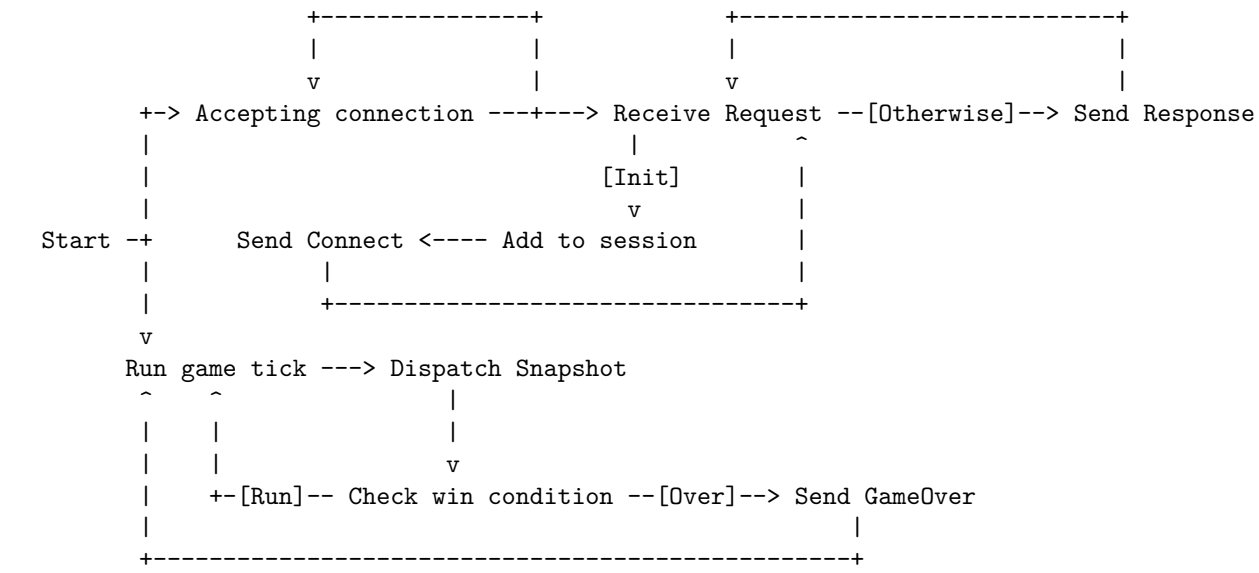
In principle, all the client has to do is:

1. Establish a connection to the server.
2. Send a single `Init` request.
3. Render the snapshot included in the `Connect` response and all future `Snapshot` events.
4. Continually send `Break`/`Throw`/`Move` actions based on player input.

Advanced clients, such as the reference implementation in this repository may choose to do some client side interpolation.

State Diagram

Server



Client