R-Type Working Group E. Aguado
Request for Comments: 1 Epitech
Category: Standards Track December 2016

Data Transport Protocol over UDP optimized for Video Games

Status of this Memo

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html

This Internet-Draft will expire on June 2017.

Copyright Notice

Abstract

Real time video games usually require an optimized network protocol in order to provide a fluid experience to the user. This document provides an example of such a protocol. Provided code examples are written in C++. This document is by no mean a BCP although some of the concepts are widely accepted among the Internet Community as good practices. This paper assumes knowledge of the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP).

Table of Contents

1. Introduction

This document describes a protocol used in a multiplayer R-Type like video game where all the authority is held by the server. All clients then act like “streamers”, meaning they only receive payloads from the server and change the display accordingly.

The nature of a video game protocol dictates a few needs that are fulfilled with UDP and hand-made serialization in this case.

2. Conventions used in this document

In examples, "C:" and "S:" indicate lines sent by the client and server respectively.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

In this document, these words will appear with that interpretation only when in ALL CAPS. Lower case uses of these words are not to be interpreted as carrying significance described in RFC 2119.

3. UDP

The protocol uses mainly UDP for its many strengths detailed below. However some parts may benefit more from a TCP protocol.

A UDP protocol is especially relevant in the gameplay part of the game more than everything else.

3.1. Reasons to pick UDP

The choice for a UDP protocol is the response to a few needs that a video game has. We take advantage of both the pros and the cons.

3.1.1. Latency or 'ping'

A key concept in an online multiplayer video game is the latency or ping. It represents the time needed for a byte to travel from the client to the server and come back. The ping is extremely important and greatly define the playability of a game. For example, a ping of 200ms means that whenever the user presses a key, the action related to that key will happen 200 milliseconds later. So the less the better.

The ping can be altered by the network quality (bandwidth) and the packet size (a big packet will take longer to transit than a small one).

UDP helps solving this problem by:

1. Having a reduced header size. TCP header is 20 bytes (2. RFC 6691) against 8 bytes for UDP (RFC 768).

2. Not acknowledging the request. UDP doesn't verify that the packet was actually received. This has the side effect that a packet can be lost for ever, but also reduces the amount of data to send.

3. Packets may be received out of order. We bypass this problem by sending small packets everytime. Theses packets' length will be lower than the network Maximum Transmission Unit.

3.1.2. Packet loss

In this very case, a packet loss is not a big problem at all. For realtime game data like a player input or the game state, only the most recent data is relevant. An older state is irrelevant since it doesn't apply anymore. This is why the lack of acknowledge is not a problem here.

3.2. Sent data

As stated, UDP gives us a lot of benefits for the gameplay aspect. However in other cases it may not be appropriate. The following chart describes exchanges between server and clients during a game session.

```
                 +--------+
            +--> | SERVER | <---+
            |    +--------+     |
            |         |         |
            |         |         |
            |         |         |
            |      Entities     |
          input       |       input
            |         |         |
            |         |         |
            |         |         |
            |         |         |
       +--------+     |     +--------+
       | CLIENT |  <--+--> | CLIENT |
       +--------+           +--------+
```

Figure 1 Exchange between clients and server

Every exchange MUST be handled asynchronously.

4. TCP

The TCP application is necessary whenever the server needs to send informations to one or every client and be sure that the transfer completes. An example would be a game lobby, where connections betweens clients are established, level is chosen, game rules are set, etc.

5. The Protocol

Data is sent at a very fast rate over the network. Depending on the implementation, we could send over 60 times the whole game state per second. Therefore, each packet size SHOULD be as lightweight as possible.

5.1. Binary data transfer

Conventional web formats (XML and JSON) are not relevant in this case. The amount of meta-data needed is problematic in terms of network performances. For XML files, describing the data alone can be heavier than the data itself. JSON, although better than XML, is still not ideal.

We can take advantage of the fact that the client and the server are developed at the same time and in close synchronization to establish a shared protocol whose rules set is known in before hand by both parties. So for the hypothetical following C++ structure :

```
struct Entity {
   uint16_t a = 42;
   char     b = 'a';
}
```

An XML encoding would look like this :

```
<Entity>
   <uint32_t name=a>42</uint32_t>
   <char name=b>a</char>
</Entity>
```

68 bytes per structure.

A JSON encoding would look like this :

```
{
   uint32: {
      name: a,
      value: 42
   },
   char: {
      name: b,
      value: a
}
```

53 bytes per structure (although we could get up to 33 bytes by using one byte per name and removing all spaces).

Now a binary format would look like this (in a big endian system) :

```
+----------+--------------------------------------------+
|     B    |                      A                     |
+----------+--------------------------------------------+
| 01000010 | 00000000 | 00000000 | 00000000 | 00101010 |
+----------+----------+----------+----------+----------+
```

With a total length of 5 bytes.

5.2. Packet structure

Once again, the tiniest the packet sent on the network is, the better. Since both parties know exactly what to expect, we can get rid of most of the meta-data. We are then left with a few mandatory specifications.

A packet sent to any party MUST comply to the following schema and MUST respect this exact order :

```
+-------+-------------+-----------------+
| CRC32 | MessageType | Message Content |
+-------+-------------+-----------------+
```

Figure 2 Packet structure

CRC32 (4 bytes): Checksum used to verify the integrity of the message. For more informations see RFC 3385.

5.3. Serializing and De-serializing

There are two main rules that are enforced in the current serializer implementation. The first one being the serialization order, and the second one is bit packing.

It is REQUIRED to ensure a synchronized order of serialization and deserialization, otherwise the client and server SHALL NOT work at all. Since we don't use any meta-data, every party must be aware of the order in which every entity is serialized.

Bit packing however is OPTIONAL. It is a good optimization and MAY be implemented to optimize further packet size.

5.3.1. Serialization order

This safety is guaranteed by having both the serializer and the de-serializer use the same code with different effects depending on the caller. The serialization is handled by a “Packer” object whose constructor follows this signature :

```
Packer(SerializationType);
```

where SerializationType is an enumeration that can take the following values :

```
+-------+-----------------+
| NAME  |      EFFECT     |
+-------+-----------------|
| WRITE |  Serialization  |
+-------+-----------------|
| READ  | Deserialization |
+-------+-----------------+
```

Figure 3 Serializer Enumerations

5.3.2. Bit packing

This optimization is OPTIONAL although efficient to optimize packet length. Bit packing consists of using only the necessary bits when serializing. This implies that the value ranges are known beforehand by both parties.

In order to implement a bit packing, serialized entities MUST provide a minimum and maximum value.

For example, considering the following C++ code consisting of variables that are meant to be serialized and sent over the network :

```
uint32_t a = 10;  // min 0, max 40
uint32_t b = 3; // min 0, max 5
```

Serializing these two variables in binary format without bit packing would take 8 bytes total. Notice how we are aware of the range of the values.

So here “a” only needs at most 6 bits and “b” only needs 3. Only 9 bits would be used out of the 64 available. Here is what the data would look like after using a bit packer :

```
              |  b  |     a     | Variable names |
--------------------------------+----------------|
              |     |           | Representation |
 0 0 0 0 0 0 0|0 1 1|0 0 1 0 1 0|       in       |
              |     |           |     Memory     |
--------------------------------+----------------|
              |  3  |     10    | Size in bits   |
--------------------------------+----------------|
             16                 |  Total size    |
```

Figure 4 Bit Packing in action

At deserialization, you will read the max number of used bits and fill the rest with zeros.

Using this method, we effectively save 6 bytes reducing the total payload size to 2 bytes.

6. Security Considerations

6.1. Payload integrity

As in every client/server situation, one should never blindly trust informations sent from a client. Many ways are available to an attacker to intercept a packet, tweak it at his will, and send it back to the server, resulting in corrupted data.

6.1.1. Separation of concerns

Messages (read “data sent from a party and interpreted”) are dispatched to every entity that would be affected and no more. So a message about the game network status would not be forwarded to any, let's say, monster or player.

The message type (the byte right after the CRC32) plays a first layer of security by preventing to spray corrupted messages to everyone.

6.1.2. CRC32

For more detailed informations about Cyclic Redundancy Check, see RFC 3385.

A CRC is a checksum used to ensure that the packet was not modified. If any part of the payload is modified, this checksum will allow us to flag it as corrupted and drop it.

6.2. Server authority

As in every client/server situation, one should never blindly trust informations sent from a client. Many ways are available to an attacker to intercept a packet, tweak it at his will, and send it back to the server, resulting in corrupted data.

We solve this problem by giving full authority to the server, and almost none to the client. A client can only do the following actions :

- o Send a message about a movement

- o Send a message about a projectile fired

Note that even if a client sends one of these events to the server, it is not guaranteed to be taken into account. If for example, a player happen to die the instant he sends a movement event, the server will no accept the event, and will notify the client that he didn't move.

This way, even if a packet get to be corrupted, it will only affect the “cheating” client and no one else.

7. Conclusions

We built this protocol around security and playability. Our main two requirements were to build a protocol that was cheat-proof and still allowed a player to experience the most lag-free experience.

8. References

[1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[2] Crocker, D. and Overell, P.(Editors), "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, Internet Mail Consortium and Demon Internet Ltd., November 1997.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2234] Crocker, D. and Overell, P.(Editors), "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, Internet Mail Consortium and Demon Internet Ltd., November 1997.

[3] Faber, T., Touch, J. and W. Yue, "The TIME-WAIT state in TCP and Its Effect on Busy Servers", Proc. Infocom 1999 pp. 1573-1583.

[Fab1999] Faber, T., Touch, J. and W. Yue, "The TIME-WAIT state in TCP and Its Effect on Busy Servers", Proc. Infocom 1999 pp. 1573-1583.

Authors’ Addresses

Enzo Aguado
Epitech
12 rue Sainte Catherine 69001 Lyon

Phone: 07 82 71 81 14
Email: enzo.aguado@epitech.eu

Etienne Pasteur
Epitech
90 boulevard Marius Vivier Merles 69003

Phone: 07 70 76 81 40
Email: etienne.pasteur@epitech.eu

Hippolyte Barraud
Epitech
16 quai Joseph Gilet 69004 Lyon

Phone: 06 47 97 25 52
Email: hippolyte.barraud@epitech.eu