Riptide Networking

Notes

February 27, 2023

Contents

1 Messages

1.1 Header

Name	Type	Comment
Header	1 Byte	Header byte
Sequence ID	2 Byte	Optional, only for reliable messages
Message ID	UShort	Only present in user defined messages

1.1.1 Header Byte

Name	Value	Description
Unreliable	0	An unreliable user message **Unreliable**
Ack	1	An internal unreliable ack message **Unreliable**
AckExtra	2	An internal unreliable ack message, used when acknowledging a sequence ID other than the last received one **Unreliable**
Connect	3	An internal unreliable connect message **Unreliable**
Reject	4	An internal unreliable connection rejection message. **Unreliable**
Heartbeat	5	An internal unreliable heartbeat message. **Unreliable**
Disconnect	6	An internal unreliable disconnect message. **Unreliable**
Reliable	7	A reliable user message. **Reliable**
Welcome	8	An internal reliable welcome message. **Reliable**
ClientConnected	9	An internal reliable client connected message. **Reliable**

ClientDisconnecte	10	An internal reliable client disconnected message. **Reliable**

`Runtime/Core/Transport/IPeer.cs`
Maximum size of message Body defaults to **1225** Bytes (`Runtime/Core/Message.cs:30`)

1.2 Value Encoding

1.2.1 Byte

Unsigned 8 bit integer

Single

Name	Type	Description
value	`byte`	Single byte, without any further processing

Array

Name	Type	Description
Array Length	`byte/ushort`	**OPTIONAL** See Array Length for details
value	`byte[]`	Byte array with a maximum length of 2^{16}. Directly copied into message body.

1.2.2 Signed Byte

Signed 8 bit integer

Single

Name	Type	Description
value	`sbyte`	Single signed byte, cast to two's complement encoded unsigned byte

Array

Name	Type	Description
Array Length	`byte/ushort`	**OPTIONAL** See Array Length for details

value	`sbyte[]`	SByte array, no maximum length. Cast to byte and copied into array one by one

1.2.3 Boolean

Single

Name	Type	Description
value	`bool`	Single boolean, encoded as single byte, with value 0x01 for `true` or value 0x00 for `false`. Values other than 0x01 will be interpreted as `false` when reading the message.

Array

Name	Type	Description
Array Length	`byte/ushort`	**OPTIONAL** See Array Length for details. **Beware:** The Array length counts the number of boolean objects in the original Array. This is not equal to the number of bytes used for storing the Array
value	`bool[]`	bool array, no maximum length. Booleans are packet into bytes (8 booleans per byte). That means the first bool is represented as the lowest bit of the first byte, the second is the second lowest bit and so on

1.2.4 Short

16 Bit signed integer

Single

Name	Type	Description
value	`short`	short, taking 2 bytes in the message. **Endianness is dependent on the host system's .net implementation. Assume Little Endian per default**

Array

Name	Type	Description
Array Length	byte/ushort	**OPTIONAL** See Array Length for details
value	short[]	Shorts added sequentially using the method for adding single shorts. **Endianness is dependent on the host system's .net implementation. Assume Little Endian per default**

1.2.5 UShort

16 Bit unsigned integer

Single

Name	Type	Description
value	ushort	ushort, taking 2 bytes in the message. **Endianness is dependent on the host system's .net implementation. Assume Little Endian per default**

Array

Name	Type	Description
Array Length	byte/ushort	**OPTIONAL** See Array Length for details
value	ushort[]	UShorts added sequentially using the method for adding single ushorts. **Endianness is dependent on the host system's .net implementation. Assume Little Endian per default**

1.2.6 Int

32 Bit signed integer

Single

Name	Type	Description

value	`int`	int, taking 4 bytes in the message. **Endianness is dependent on the host system's .net implementation. Assume Little Endian per default**

Array

Name	Type	Description
Array Length	`byte/ushort`	**OPTIONAL** See Array Length for details
value	`int[]`	Integers added sequentially using the method for adding single ints. **Endianness is dependent on the host system's .net implementation. Assume Little Endian per default**

1.2.7 UInt

32 Bit unsigned integer

Single

Name	Type	Description
value	`uint`	uint, taking 4 bytes in the message. **Endianness is dependent on the host system's .net implementation. Assume Little Endian per default**

Array

Name	Type	Description
Array Length	`byte/ushort`	**OPTIONAL** See Array Length for details
value	`uint[]`	Integers added sequentially using the method for adding single uints. **Endianness is dependent on the host system's .net implementation. Assume Little Endian per default**

1.2.8 Long

64 Bit signed integer

Single

Name	Type	Description
value	`long`	long, taking 8 bytes in the message. **Endianness is dependent on the host system's .net implementation. Assume Little Endian per default**

Array

Name	Type	Description
Array Length	`byte/ushort`	**OPTIONAL** See Array Length for details
value	`long[]`	Integers added sequentially using the method for adding single longs. **Endianness is dependent on the host system's .net implementation. Assume Little Endian per default**

1.2.9 ULong

64 Bit unsigned integer

Single

Name	Type	Description
value	`ulong`	ulong, taking 8 bytes in the message. **Endianness is dependent on the host system's .net implementation. Assume Little Endian per default**

Array

Name	Type	Description
Array Length	`byte/ushort`	**OPTIONAL** See Array Length for details
value	`ulong[]`	Integers added sequentially using the method for adding single ulongs. **Endianness is dependent on the host system's .net implementation. Assume Little Endian per default**

1.2.10 Float

32 Bit signed IEEE floating point

Single

Name	Type	Description
value	`float`	float, taking 4 bytes in the message, Encoded in IEEE format, seperated into its single bytes. **Endianness is dependent on the host system's .net implementation. Assume Little Endian per default**

Array

Name	Type	Description
Array Length	`byte/ushort`	**OPTIONAL** See Array Length for details
value	`float[]`	Floats added sequentially using the method for adding single IEEE floats. **Endianness is dependent on the host system's .net implementation. Assume Little Endian per default**

1.2.11 Double

64 Bit signed IEEE floating point

Single

Name	Type	Description
value	`double`	double, taking 8 bytes in the message, Encoded in IEEE format, seperated into its single bytes. **Endianness is dependent on the host system's .net implementation. Assume Little Endian per default**

Array

Name	Type	Description
Array Length	`byte/ushort`	**OPTIONAL** See Array Length for details

value	`double[]`	Floats added sequentially using the method for adding single IEEE doubles. **Endianness is dependent on the host system's .net implementation. Assume Little Endian per default**

1.2.12 String

UTF-8 Encoded String

Single

Name	Type	Description
length	`byte/ushort`	Length of the encoded byte array
value	`byte[]`	UTF-8 encoded string

Array

Name	Type	Description
Array Length	`byte/ushort`	**OPTIONAL** See Array Length for details
value	`string[]`	Strings added sequentially using the method for adding single Strings.

1.2.13 Vector2

Name	Type	Description
value.x	`float`	x component of Vector2
value.y	`float`	y component of Vector2

1.2.14 Vector3

Name	Type	Description
value.x	`float`	x component of Vector3
value.y	`float`	y component of Vector3
value.z	`float`	z component of Vector3

1.2.15 Quaternion

Name	Type	Description

value.x	`float`	x component of Quaternion
value.y	`float`	y component of Quaternion
value.z	`float`	z component of Quaternion
value.w	`float`	w component of Quaternion

1.2.16 IMessageSerializable

Custom Structures

Single

Name	Type	Description
value	`IMessage-Serializable`	Serialized using Serialize() method of this Object, deserialized using Deserialize() method of the type. Expected type has to be declared by user, Type must have no-parameter constructor

Array

Name	Type	Description
Array Length	`byte/ushort`	**OPTIONAL** See Array Length for details
value	`IMessage-Serializable[]`	IMessageSerializables added sequentially using the method for adding single IMessage-Serializables.

1.2.17 Array Length

Supports array lengths up to 32767 elements. Larger arrays are not supported.
The actual serialisation of the array length depents on the value serialized. If the value is less than 127, a single byte is used. Otherwise, two bytes are used.

Values $\leq$ 127

Name	Type	Description
value	`byte`	Length serialized in a single byte with the highest bit set to 0

Values $>$ 127 and $\leq$ 32767

Name	Type	Description
value	byte	High byte of the length, with the highest bit set to 1 in order to indicate 2 bytes used for the array length
value	byte	Lower byte of the length

Values > 32767

Throws argument out of range exception.

1.3 Message Types

1.3.1 Unreliable

An unreliable user message

Name	Type	Description
Message Type	byte	Value set to 0
Message ID	ushort	Message ID
payload	byte[]	Payload defined by user, with data types serialized as described in Value Encoding

1.3.2 Ack

An internal unreliable ack message

Name	Type	Description
Message Type	byte	Value set to 1
LastReceived-SeqId	ushort	Last remote sequence ID
AcksBitfield	ushort	Acks (binary flags)

1.3.3 AckExtra

An internal unreliable ack message, used when acknowledging a sequence ID other than the last received one

Name	Type	Description
Message Type	byte	Value set to 2
LastReceived-SeqId	ushort	Last remote sequence ID
AcksBitfield	ushort	Acks (binary flags)

forSeqId	ushort	Sequence ID this ack is for

1.3.4 Connect

An internal unreliable connect message

Name	Type	Description
Message Type	byte	Value set to 3
connectBytes	byte[]	**OPTIONAL** Custom data to include when connecting. **Length of the Array is not included in the message**

1.3.5 Reject

An internal unreliable connection rejection message

Name	Type	Description
Message Type	byte	Value set to 4
RejectReason	byte	Reason for the rejection of the connection. See also Reject Reasons
rejectMessage	byte[]	**OPTIONAL** custom byte[] containing additional data. See Value Encoding for value. Length of the array is not included. If this field is present, **RejectReason must be set to Custom**

1.3.6 Heartbeat

An internal unreliable heartbeat message

Name	Type	Description
Message Type	byte	Value set to 5
Ping ID	byte	Ping ID of the message
RTT	short	Round trip time, -1 if not calculated jet

1.3.7 Disconnect

An internal unreliable disconnect message

Name	Type	Description
Message Type	byte	Value set to 6
Reason	byte	Disconnect reason, see also Disconnect Reasons

Message	byte[]	**OPTIONAL** custom byte[] containing additional data. See Value Encoding for value. Length of the array is not included. If this field is present, **Disconnect Reason must be set to Kicked**

1.3.8 Reliable

A reliable user message

Name	Type	Description
Message Type	byte	Value set to 7
Sequence ID	ushort	Sequence ID
Message ID	ushort	Message ID
payload	byte[]	Payload defined by user, with data types serialized as described in Value Encoding

1.3.9 Welcome

An internal reliable welcome message

Name	Type	Description
Message Type	byte	Value set to 8
Sequence ID	ushort	Sequence ID
ID	ushort	Connection ID

1.3.10 ClientConnected

An internal reliable client connected message. Send to all clients when a new client connects.

Name	Type	Description
Message Type	byte	Value set to 9
Sequence ID	ushort	Sequence ID
ID	ushort	Client ID

1.3.11 ClientDisconnected

An internal reliable client disconnected message. Send to all still connected clients when a client disconnects.

Name	Type	Description
Message Type	byte	Value set to 10

Sequence ID	ushort	Sequence ID
ID	ushort	Client ID

1.4 Enums

1.4.1 Reject Reasons

See `Core/Peer.cs`

Name	Value	Description
0	NoConnection	No response was received from the server (because the client has no internet connection, the server is offline, no server is listening on the target endpoint, etc.).
1	AlreadyConnected	The client is already connected.
2	Pending	A connection attempt is already pending.
3	ServerFull	The server is full.
4	Rejected	The connection attempt was rejected.
5	Custom	The connection attempt was rejected and custom data may have been included with the rejection message.

1.4.2 Disconnect Reasons

See `Core/Peer.cs`

Name	Value	Description
0	NeverConnected	No connection was ever established
1	ConnectionRejected	The connection attempt was rejected by the server
2	TransportError	The active transport detected a problem with the connection
3	TimedOut	The connection timed out or the real reason for the disconnect was lost / is unclear
4	Kicked	The client was forcibly disconnected by the server
5	ServerStopped	The server shut down
6	Disconnected	The disconnection was initiated by the client

1.5 Conclusions from Message Implementation

- Endianness depends on the local hardware. This may lead to values not being interpreted correctly if server and client use different endianness. Assume Little-Endian data as the default case.
- For (send-)performance, prefer byte[] arrays over sbyte[] arrays.
- *If byte[] arrays with more than 2^{16} elements need to be transferred, use sbyte[] instead, otherwise there will be an overflow in the write pointer of the message class.*
  - Fixed with commit ba93196, 12.02.2023
- It seems possible to create messages that are up to 5 byte larger than the expected package size
- It seems possible to accidentally get crappy data by passing an oversized array to GetBools to read into
- Using getBool on Bool arrays will fail.
- TCP Transport seems to listen mostly on IPv6 (IPv6Any in C#)
- It is possible to try to send negative amount of data, causing generic Argument out of range exceptions using TCP transport
- It is possible to spoof TCP messages by sending carefully crafted messages with a length of $> 2^{16}$ bytes
- Docs in client.cs:312 (local disconnect, rejectReason) seems wrong (copy-paste error)

2 Protocol

2.1 General Information

- **HeartbeatInterval**: 1000 ms
- **Timeout**: 5000 ms
- **Resend Interval**: 1.2 * smoothRTT
- **Resend Attempts**: 15

2.2 Connection

1. Client initiates connection from Transport and Client starts heartbeat messages
2. The server either
   - accepts the connection (if no custom connection handler is specified)
   - adds the connection to the list of pending connections, if a custom connection handler is specified & calls the handler.
3. Once the connection is accepted, a welcome message is sent from the server to the client

2.3 Heartbeat

The heartbeat is used in order to check if connections are timed out, and to measure the round trip time for packets

2.3.1 Client

The heartbeat is started once the connection is initiated (by calling the connect method). The behavior of the heartbeat depends on the current state of the client.

isConnecting

If the maximum connect attempts are not reached, sends a connect message to the remote peer. If connectBytes is not Null, the connect bytes are appended. The connection attempt counter gets incremented. Otherwise, a local disconnect is called with reason "NeverConnected"
Finally, the next heartbeat event is scheduled.

isPending

If the current connection attempt timed out, a local disconnect is called with reason "TimedOut"
Otherwise the next heartbeat event is scheduled.

isConnected

If the current connection timed out, a local disconnect is called with reason "TimedOut". Otherwise a heartbeat is sent and the next heartbeat event is scheduled.

Other states

The next heartbeat event is scheduled.

2.3.2 Server

The heartbeat is started once the server starts.

2.4 Reliable Messages

Reliable messages are messages that need to be acknoledged by the peer. If within 1.2 times the current smoothed round trip time (as determined by the heartbeat) no ack is received, the message is resent. If more than 15 attempts are tried, the message gets discarded, and a warning gets logged, if enabled.

2.4.1 Duplicate Detection

See also `Core/Connection.cs:141`
First the gap between the received sequence ID and the previously received sequence ID is computed. The next steps depend on the sign of the gap:

Positive Gap (larger sequence ID received)

Once a reliable message with an newer sequence ID than the previous (= positive sequence Gap) is received, an 64 bit long bit field is shifted left by the gap since the last received sequence ID.

if gap $\leq$ 16: the acks bitfield gets shiftet by sequence bits. The new acks bitfield now consists of the two most minor bytes of the shifted value, the "Overflow" gets then or-ed into the duplicate filter bitfield
The message is handled if the bit in the acks bit field at position sequenceGap is zero. When handling, this bit is flipped to one.
The last received sequence ID is set to this message's sequence ID

if gap $\leq$ 80: Shifts the acks bit field by sequenceGap-16. The shifted bits are or-ed into the duplicate filter bitfield, and the acks bit field is zeroed
The message is handled if the bit in the duplicate bit field at position sequenceGap-16 is zero. When handling, this bit is flipped to one.
The last received sequence ID is set to this message's sequence ID

Negative Gap (smaller sequence ID received)

if gap $\leq$ 16: The message is handled if the bit in the acks bit field at position abs(sequenceGap) is zero. When handling, this bit is flipped to one.

if gap $\leq$ 80: The message is handled if the bit in the duplicate bit field at position abs(sequenceGap) is zero. When handling, this bit is flipped to one.

Gap of 0

Is not handled.

Finally, an Ack message is sent for the sequence ID

2.4.2 Acknoledge Messages

See also `Core/Connection.cs:233`
First the gap between the received sequence ID and the previously received sequence ID is computed. The next steps depend on the sign of the gap:

Positive Gap (larger sequence ID received)

For each id in the gap, excluding the current message, first the acked messages bit field is shifted left by one. Then, the left most message is checked for ack status. If the message is already acked, the pending message gets cleared from the pending messages dict, if still present. If no ack for the message has been received yet, the message is resent if present in the pending messages dict.
Once all previous messages from the acked messages bit field are checked, the bit field gets shifted once more left to make space for the ack bit of the current sequenceID. The ackedMessagesBitfield is then or-ed with the remote acks bitfield from the ack message and the ack bit for the current sequence ID.
The lastAckedSeqID is then set to the remote last received SeqID

Negative Gap (smaller sequence ID received)

According to comments in the source, this branch most likely never executes.
The bit corresponding to the sequence ID is set, and the local acked bitfield is or-ed with the remote acksBitField, ensuring that the bit corresponding to this ack is set to 1.
If there exists still a pending message for this ack, the pending message is cleared.

Gap of 0

The remote and local bit fields are combined (using binary or), and the ack status of the oldest sequence ID is checked.

2.5 Transport Details

2.5.1 UDP

If not otherwise specified, UDP is used as transport.

2.5.2 TCP

Package Format

Name	Type	Description
packetLength	`ushort`	Length of the package
packet	`Message`	Content of the package (= the Message)

Please note that the byteOrder of packetLength is again most likely system dependent. Expect Little Endian as default. (if the Symbol "BIG_ENDIAN" is not defined)

3 Events

3.1 Transport

3.1.1 IPeer

DataReceived

Name	Type	Description
dataBuffer	`byte[]`	An array containing the received data
amount	`int`	The number of bytes that were received
fromConnectior	`Connection`	The connection which the data was received from

Disconnected

Name	Type	Description
connection	`Connection`	The connection which was closed
reason	`DisconnectReaso`	The reason for the disconnection

3.1.2 IClient

Inherits events from IPeer

Connected

No Arguments

ConnectionFailed

No Arguments

3.1.3 IServer

Inherits events from IPeer

Connected

Name	Type	Description
connection	Connection	Connection that just got connected

3.2 Interface

3.2.1 Client

Connected

No Arguments

ConnectionFailed

Name	Type	Description
message	Message	Additional data related to the failed connection attempt (if any)

MessageReceived

Name	Type	Description
fromConnectior	Connection	The connection from which the message was received
messageID	ushort	ID of the Message
message	Message	the received Message

Disconnected

Name	Type	Description
reason	DisconnectReaso	The reason for the disconnection
message	Message	additional data related to the disconnection (may be null)

ClientConnected

Name	Type	Description
id	ushort	The numeric ID of the client that connected

ClientDisconnected

Name	Type	Description
id	ushort	The numeric ID of the client that disconnected

3.2.2 Server

ClientConnected

Name	Type	Description
client	Connection	The newly connected client

MessageReceived

Name	Type	Description
fromConnectior	Connection	The connection from which the message was received
messageID	ushort	ID of the Message
message	Message	the received Message

ClientDisconnected

Name	Type	Description
client	Connection	The client that disconnected
reason	DisconnectReaso	The reason for disconnection

4 Usage

4.1 C#

Based on `https://riptide.tomweiland.net/manual/overview/getting-started.html`

4.1.1 Initialize Logging

```
RiptideLogger.Initialize(Debug.Log, Debug.Log, Debug.LogWarning, Debug.
    LogError, false);
```

4.1.2 Create Server

```
Server server = new Server();
server.Start(7777, 10);
```

In order to process the messages:

```
private void FixedUpdate()
{
  server.Update();
}
```

4.1.3 Create Client

```
Client client = new Client();
client.Connect("127.0.0.1:7777");
```

In order to process the messages:

```
private void FixedUpdate()
{
  client.Update();
}
```

4.1.4 Messages

```
Message message = Message.Create(MessageSendMode.Unreliable, 1);
```

4.2 Python

4.2.1 Create Server

```
    tcpTransport = TCPServer()
    server: Server = Server(tcpTransport)
    server.start(PORT, 10)
```

In order to process the messages:

```
    serverUpdater: FixedUpdateThread = FixedUpdateThread(server.update)
    serverUpdater.start()
```

4.2.2 Create Client

```
    tcpTransport = TCPClient()
    client: Client = Client(tcpTransport)
    client.connect(("127.0.0.1", PORT))
```

In order to process the messages:

```
    clientUpdater: FixedUpdateThread = FixedUpdateThread(client.update)
    clientUpdater.start()
```

4.2.3 Messages

```
    msg = message.create(MessageSendMode.Unreliable, MESSAGE_ID_HANDLED)
    msg.putString("Hello World !")
    client.send(msg)
```