# XNA Network Protocol Specification, Draft 2

Brady Dial, Charles McGarvey

March 7, 2010

## Contents

1 Rationale ........................................ 2

2 Overview ....................................... 2

3 The Lobby ...................................... 3

4 In-Game ........................................ 5
   4.1 Updating Under Good Conditions .......... 5
   4.2 Latency Readjustment ..................... 5
   4.3 Handling Stalls and Dropping Clients ..... 6

5 Packets ......................................... 10
   5.1 Chat ....................................... 10
   5.2 Event ..................................... 10
   5.3 Stall ...................................... 12

A Changes/Improvements ......................... 13

B Limitations ................................... 14

C Future Development ................................ 14
1 Rationale

This document describes a network topology and protocol designed for XNA games. This specification was created with the goals of minimum latency and maximum flexibility.

2 Overview

The protocol assumes a peer-to-peer topology where each client sends packets directly to each other client. The basic strategy specified is that of beginning each game with a known state and communicating keyboard and mouse events between all of the clients, and allowing the game state to update itself deterministically.

The specification also describes a mechanism to allow the latency to be adjusted on-the-fly to adapt to changing network conditions. The average latency should be slightly above the OWD\(^1\) of the worst route between clients.

The convention used in this specification is to use number of frames\(^2\) to describe latency. For example, at 60Hz the time between each frame is 16.67ms, so a latency of 200ms is 12 frames. This is only an example; the protocol itself will work with any frame-rate. Each individual frame must be numbered, each consecutive frame identified by a number one greater than the last and the first frame being zero.

For a summary of the changes made for this draft, see Appendix A.

---

\(^1\) One-Way Delay is the time it takes one packet to be transmitted from one client to another.

\(^2\) A frame is one update/draw cycle. The update and draw need not be coupled, but it is convenient to think about the ratio as 1:1 update and draw. If decoupled, frame refers to a complete update where events are applied and state is changed.
3 The Lobby

XNA provides most of the necessary functionality for transferring players from the lobby to the game and vice versa. The protocol specified here includes support for chat messages to be sent between clients in the lobby. See Figure 2 for a visualization of the activities performed in the lobby update.

Figure 1: This diagram shows that a different path is followed for updating, depending on whether a game is in-progress or not.
Lobby Update

- [client is XNA host]
  - [else]
    - [all players are ready]
      - Send and receive chat packets
      - Signal XNA to begin the game

- [host started game]
  - Game Begins

- [else]

Updates will now go through the in-game update. This marks a state transition.

Game Begins

- Set latency to 1 frame
- Calculate average OWD
- Set frame counter to zero
- Set stall counter to zero

Initialize the state information we need for the in-game update.

Figure 2: Apart from the lobby functionality already provided by XNA, the lobby update only consists of all the clients passing chat packets between each other client in the same network session. Additionally, the XNA host also has the responsibility to signal the game to begin.
4  In-Game

The in-game activities and interactions are a bit more complex than they are in the lobby. They will be specified here, but readers should study Figures 3, 4 and 5 for complete understanding. Also, make sure you understand the structure and fields of the event packet; see Table 6. Beware, a lot of text follows! In case you’re still reading this paragraph, I highly recommend you visit the diagrams and packet tables mentioned and only read this stuff for clarification in case the diagrams themselves are lacking. Now we begin.

For each frame, each client shall first check for and parse any received event packets. The packet should contain events which ought to be applied at a future frame, so the client should store events according to when they should be applied, taking note of the player who sent the events.

4.1 Updating Under Good Conditions

If the client has received all event information from each of the clients for the current frame, the client will send an event packet, apply the events for the current frame, update game state and increment the frame counter. The client might also have to readjust the latency if the current frame was previously\(^3\) determined to be when the latency should be readjusted; after the latency adjustment, the stall counter should be reset to zero. See Figure 3 to see when a readjustment should occur in relation to other activities.

4.2 Latency Readjustment

The latency readjustment algorithm uses the stall counter and average OWD from each client as it was before the last readjustment. If the maximum stall counter among all the clients is more than zero (i.e. at least one of the clients stalled for at least one frame), then the latency is increased by that number of frames. Otherwise, the latency is decreased by 60% of the difference between the current latency and the maximum average OWD of all the clients, plus 1. In the case calculating a fractional latency, the result should be rounded down. After the latency readjustment, the latency may need to be constrained to a minimum of 1 frame and a maximum of 120 frames (two seconds at 60Hz). Because each client performs this same algorithm with the same metrics, each client will always readjust to the same latency. At this point, the client should determine the average OWD which it will send with each event packet during the following “cycle” until the next time the latency is readjusted. Section 5.2 contains information regarding the calculation of these metrics.

\(^3\) Each time the latency is readjusted, it is determined at what frame the next readjustment will occur. The first readjustment takes place after the first frame.
4.3 Handling Stalls and Dropping Clients

If, on the other hand, the client has not received all event information from each of the other clients for the current frame, the client shall increment a counter to keep track of the number of stalled frames. The first frame during a stall, the client will resend events reliably to the unresponsive client(s), which are the ones we have no events for the current frame. We will say that these clients are the ones being waited on. If the stall counter has passed 15s (900 frames at 60Hz), the “timeout” has been reached and the client drops every client that is determined to be waited upon by this or any other client. The client determines which clients are being waited on by receiving stall packets from other responsive clients. After the problematic client(s) are dropped, the stall counter is reset to zero, and the game will continue normally as long as each client continues to get the event information they need each frame from the remaining player(s).

As long as the stall counter is below the timeout limit, the local client will send stall packets to each other responsive client to tell them which clients are being waited on. The stall packet is sent reliably every second during the stall. The timing of this packet is not critical for correct synchronization; see Figure 3 to see how this could be accomplished. If the local client starts receiving packets from the distant client, the stall will cease and the local client will include the stall counter in event packets sent after the next latency readjustment. The stall packet is the specified method by which each client will drop the same problematic client(s), thereby reducing the chance of the connection state becoming chaotic or questionable after a drop occurs. See Section 5.3 for information about the stall packet.

That’s all there is to it! The in-game update will continue until the XNA host signals the end of the game or the game logic indicates that the game is over. Each client should be subscribed to the appropriate XNA events to make this happen. At the end of the game, the flow reverts back to the lobby. If the lobby is disbanded, it is up to the game to allow the player to join a new XNA network session.
Receive event and stall packets from other clients

[else]

[received events for this frame from every other client]

[else]

[stall counter surpasses threshold]

[stall counter is 1]

[stall counter is a multiple of 60]

[else]

[else]

[stall counter is zero]

[else]

[this frame is the latency readjustment frame]

Adjust Latency

Send local events to other clients

Apply events for this frame

Update game state

Increment frame counter

Increment stall counter

Send stall packet to responsive clients.

Drop all players who are being waited upon by this or any other player. Stall packet will inform other clients who is waiting on who.

Handle Stall Procedure

In-game Update

Client should store events according to the future frame number when they should be applied and ignore events for frames that are past.

[received events for this frame from every other client]

Apply events for this frame

Update game state

Increment frame counter

Figure 3: The activities performed by the in-game update.
Latency Readjustment

[else]

[any of the clients stalled last cycle]

Change latency by 60% of the difference between the current latency and the maximum average OWD, plus 1.

Increase latency by maximum number of stalled frames by any player.

Enforce lower bound of 1 frame and upper bound of 120 frames.

Set next latency readjustment frame to current frame plus the new latency.

Calculate average OWD to be sent in event packets and used during the next adjustment.

The latency readjustment algorithm is based on the average OWD, and the number of stalled frames.

Figure 4: The latency readjustment algorithm.
Each packet contains a backlog of events since the last latency readjustment.

Events Applied by All

Send Event Packet

Everyone Stalls

Send Reliable Event Packet

Send Stall Packets

Send Stall Packet

Everyone Else Resumes

If Frame Info is not received
GOTO -> Stall Sequence Diagram

Player that did not receive packet will stop sending events, causing everyone else to stall.

When Players stall they do not advance the frame count. If communication is not reestablished with player he is dropped after 15s or whatever timeout the application uses.

Figure 5: Sequence diagram showing the in-game communication between players under normal conditions as well as the interaction that occurs when a client stalls. In the stall sequence, Player 3 is shown to be missing event information from Player 1 and stalls; resending his own events and a stall packet to Players 2 and 4. The other players likewise stall as a result of Player 3 stalling, until Player 3 gets Player 1’s events and continues. Then, the other players continue after receiving Player 3’s events.
5 Packets

5.1 Chat

Chat packets shall be sent reliably and in-order from one client to other connected clients while the players are in the lobby. A chat packet need not be sent to every connected client; rather, it should only be sent to those who ought to receive the message. A chat packet may be ignored by a client which is not in the lobby. See Table 1 for the format of a chat packet.

<table>
<thead>
<tr>
<th>Data</th>
<th>Size</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte: 0x01</td>
<td>1</td>
<td>Packet ID</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>Number of characters in the message</td>
</tr>
<tr>
<td>char[]</td>
<td>variable</td>
<td>Message string as character array</td>
</tr>
</tbody>
</table>

Table 1: Format of chat packet. Sizes are in bytes.

5.2 Event

Each event packet contains an array of events to be applied at future frames. They also contain two other unrelated fields which are used as metrics for the latency readjustment algorithm. Event packets shall only be sent by clients which are involved in a game. Event packets received any other time should be ignored.

One event packet shall be sent by each client per frame with all the events that occurred since the latency was readjusted two times ago, including events for the current frame; therefore, events are sent redundantly to minimize stalls. Because of this redundancy, these packets should be sent unreliably without concern for order. See Figure 3 and Figure 5 to see how clients send and receive event packets. Event packets are formatted according to Table 6. Different events are formatted differently, according to Tables 3, 4 and 5; the data fields from these tables are “sub-packets” to be included in the array of events as shown in Table 6.

Two metrics are being piggy-backed inside the event packet. The first metric is the number of stalls the client counted during the last “cycle,” or the time before the last latency readjustment and the one prior. This is a count of the number of frames that passed while the client could not update because of a lack of event information from another client. The second metric is the average OWD that the client calculated during the last latency readjustment. The average OWD is calculated by adding up the RTT between this client and each other client and dividing by twice the number of other clients. XNA provides the means to determine the RTT between a local client and each peer. See Section 4.2 for a discussion on the latency readjustment algorithm.
Table 2: Format of event packet. Sizes are in bytes.

<table>
<thead>
<tr>
<th>Data</th>
<th>Size</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte: 0x02</td>
<td>1</td>
<td>Packet ID</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>Number of frames spent stalled during last cycle</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>Average OWD (in frames) calculated during last readjustment</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>Packet contains events up to and including this frame number</td>
</tr>
<tr>
<td>byte</td>
<td>1</td>
<td>Number of events contained in this packet (they follow)</td>
</tr>
<tr>
<td>event variable</td>
<td></td>
<td>Event 1</td>
</tr>
<tr>
<td>event variable</td>
<td></td>
<td>Event 2</td>
</tr>
<tr>
<td>event variable</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Table 3: Format of keyboard event. Sizes are in bytes.

<table>
<thead>
<tr>
<th>Data</th>
<th>Size</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte: 0x01 or 0x02</td>
<td>1</td>
<td>Event ID: key down and key up, respectively</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>Future frame to apply the event</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>Key code</td>
</tr>
</tbody>
</table>

Table 4: Format of mouse button event. Sizes are in bytes.

<table>
<thead>
<tr>
<th>Data</th>
<th>Size</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte: 0x03 or 0x04</td>
<td>1</td>
<td>Event ID: mouse down and mouse up, respectively</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>Future frame to apply the event</td>
</tr>
<tr>
<td>byte: 0x01, 0x02 or 0x03</td>
<td>1</td>
<td>Which button: left, right or middle, respectively</td>
</tr>
</tbody>
</table>

Table 5: Format of mouse motion event. Sizes are in bytes.

<table>
<thead>
<tr>
<th>Data</th>
<th>Size</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte: 0x05</td>
<td>1</td>
<td>Event ID</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>Future frame to apply the event</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>X-coordinate of cursor position</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>Y-coordinate of cursor position</td>
</tr>
</tbody>
</table>
5.3 Stall

The stall packet is sent by each client that is waiting for the events from any other client for the current frame. This is done to communicate to other responding clients which clients are being waited upon. This packet is sent at the first frame when a stall occurs and each second afterward while the stall continues. Again, this packet is only to be sent to responding clients, and the players listed within the packet are the ones you are not sending this packet to. See Section 4.3 to learn how this packet is used in the in-game update.

<table>
<thead>
<tr>
<th>Data</th>
<th>Size</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte: 0x03</td>
<td>1</td>
<td>Packet ID</td>
</tr>
<tr>
<td>byte</td>
<td>1</td>
<td>Number of clients being waited on</td>
</tr>
<tr>
<td>byte</td>
<td>1</td>
<td>Player ID</td>
</tr>
<tr>
<td>byte</td>
<td>1</td>
<td>Player ID</td>
</tr>
<tr>
<td>byte</td>
<td>1</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 6: Format of stall packet. Sizes are in bytes.
A Changes/Improvements

The following is a summary of the changes made for Draft 2:

- The begin game process was adjusted so all clients initialize data. See Figure 2.

- A "Frame to Apply to" field has been added to each event "subpacket." A field was also added to let clients know the furthest frame the events could apply to; this is needed so clients know they have the event information from a client when no event occurred during a frame. See Section 5.2.

- The latency adjustment algorithm was altered to decrease occurrences of lowering latency to the point of causing stalls. As long as the process of determining latency is followed, each client will set the next latency adjustment frame to be the same. Each client will have a list of OWDs from which they can all choose the same maximum OWD to base calculations upon. This ensures that all clients change their latency simultaneously and the latency stays synchronized between clients. See Section 4.2.

- To avoid cascading drops and to add a dropped player consensus, a stall packet was added to ensure that the cause of player stalls is known by all and only the problem clients are dropped. This also gives each player the information to know which client or clients need to be dropped due to excess stalls originating from them. Thus each player’s game will contain an accurate list of current players. See Section 5.3.

- Other minor oversights were corrected and some of the figures and descriptions were modified for improved clarity.

- Good luck! You may need it...
B Limitations

This model is not without its weaknesses. As with all things, there were tradeoffs involved in the design of the protocol. While an attempt was made to choose the best solution given the requirements and design goals, there are some inherent limitations:

- There is no mechanism to allow players to move into a game that is already in progress; the game roster is fixed once the game is started until the end or until a player is dropped or leaves. Note that the game can continue in the event of an unexpected disconnection of one or more clients.

- Latency is determined by the worst connection between any of the clients.

- Each client has all of the game state. While this significantly simplifies the interaction between the clients, this opens the door for visibility hacks.

- Protocols similar to this have been used in early FPS games which were notorious for their lousy network support. A much different protocol with a client-server topology is used with a lot of success in today’s FPS and MMORPG games. On the other hand, certain types of games (including RTS) have performed very well under protocols similar to this one.

- This model is only good provided that the original assumption of a deterministic state change holds true.

C Future Development

The specification has described a model which is fairly flexible, and it should perform well under many circumstances. The protocol could easily be extended to support game state prediction with backtracking and/or interpolation which may reduce latency further. These features were not included in the current specification because they impose extra requirements on the game implementation. Regardless, the protocol as specified is a good starting point and is quite usable as-is.