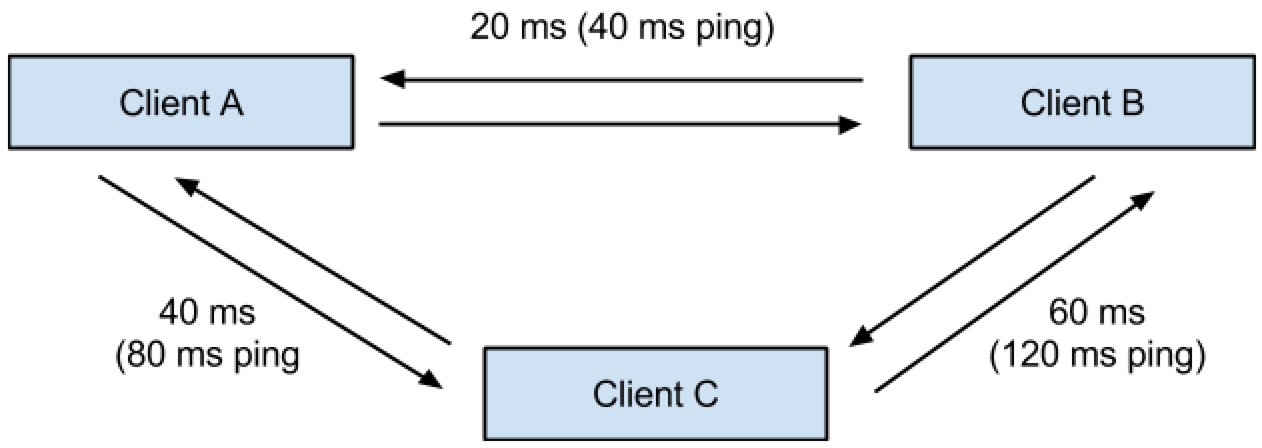
New Peer-to-Peer Netplay System for FS-UAE?

Frode Solheim, 2013-10-21

This document discusses how peer-to-peer communication can be used to provide a netplay system which less lag than a traditional client-server model.



The drawing illustrates three players involved in a peer-to-peer Amiga netplay game. Each arrow represents an UDP “connection”, with reliable in-order packet delivery implemented on top of UDP. A good choice here is to use ENet to provide the reliable UDP communication.

Three players are chosen as example here because it illustrates well how communication and synchronization between N players would work (only a two player example would not illustrate this as well).

Background

Amiga netplay in FS-UAE works by running the emulation identically for all players / clients. This requires the following (among other things):

1. All players must have the same config (at least the config keys which influences the emulation state), including the exact same game media.
2. The emulation must run in a deterministic mode to ensure that the emulation state **only** depends on the initial state + the input events. The UAE emulation code in FS-UAE is thoroughly patched to ensure this when running in netplay mode (i.e. no randomness).

3. All input events must be applied at the same emulated time, in the same order, for all clients. This, along with 1 and 2 ensures that the emulation is kept in sync for all clients.

Requirement 3 means that all clients must agree on the order of input events from all clients, and also that they must be applied at the same time for all clients (meaning at the same point in emulation, e.g. frame X on vertical position Y).

The existing netplay functionality uses a client-server model to solve requirement 3. All input events are sent to the netplay server, which orders them, and sends them back out again to all the clients with information about when to apply it (on which frame). If the server applies an input even as soon as possible, this means that client X will have an effective latency about equal to the **round-trip time** (RTT) between the client and server. The latency of client Y can be different, and will generally not affect the latency of client X. Unless the server compensates for this by introducing artificial latencies for low-latency players, the result will be a bit unfair netplay, with the lower-latency player having an advantage in a fast-paced action game.

The Peer-to-Peer Alternative

With a peer-to-peer network-based netplay game, the goal is to reduce the latency by achieving a latency near the **one-way time** instead of the **round-trip time**. I.e. in the best case, it could near halve the latency in the system which is due to network communication[1].

It is now a good time to take another look on first illustration. In that example we see that:

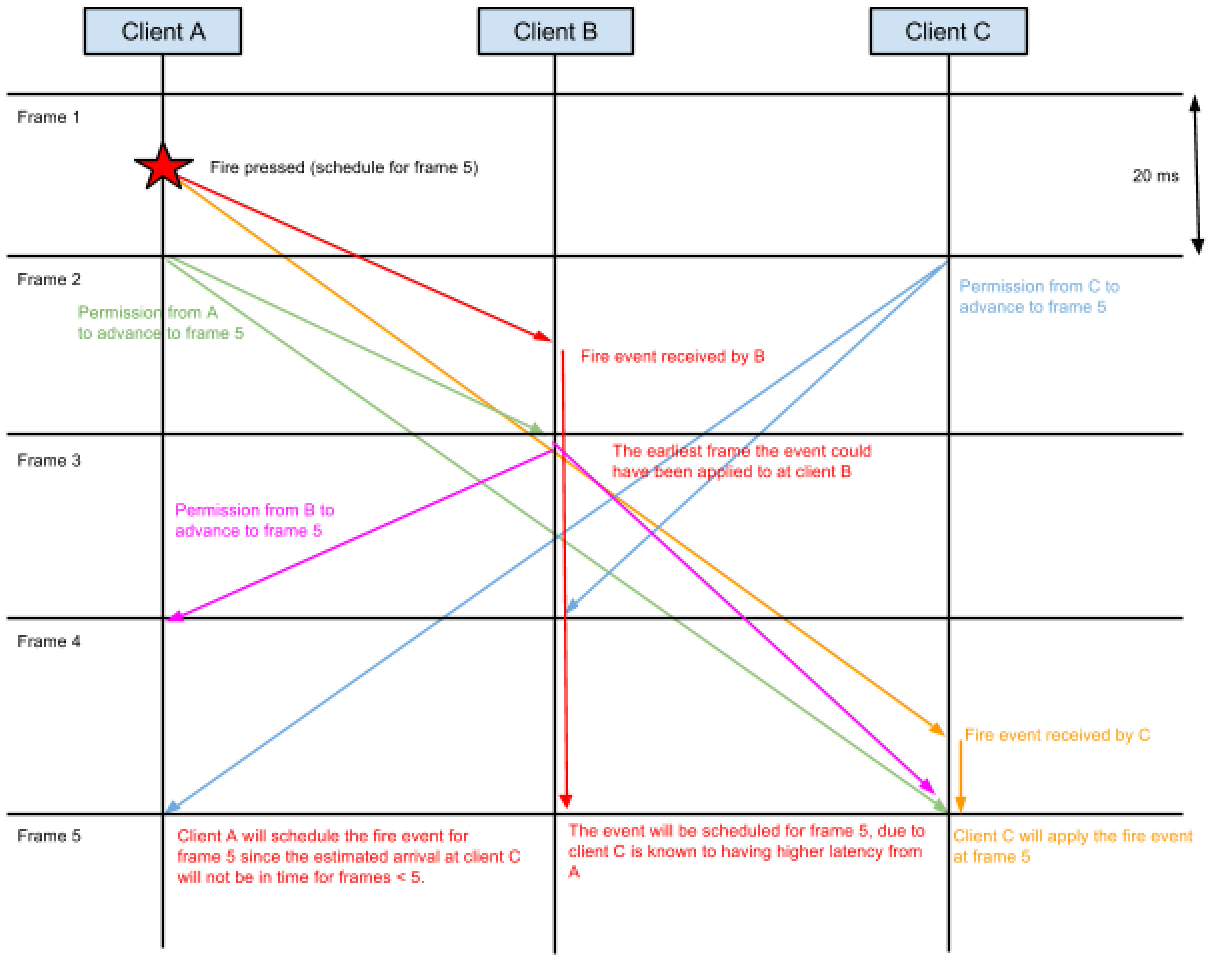
- One-way time between A and B is 20 ms (= 40 ms round-trip ping)
- One-way time between B and C is 40 ms (= 80 ms round-trip ping)
- One-way time between A and C is 60 ms (= 120 ms round-trip ping)

To further simplify the discussion of this example - and to illustrate the net play protocol, the following (overly optimistic) assumptions are made:

- The clients have complete and correct knowledge of how much time it takes to send a message to each of the other players, and we assume the network connection is perfect, and the one-way time between two specific clients are the same all the time. I.e. it is always 60 ms between A and C.
- All the clients start the emulation of a specific frame at exactly the same wall time.

[1] There are some parts of the system which introduces an certain amount of latency regardless of the netplay delay, for example input device polling delays and delays to the image is shown on screen, so halving the network latency will not necessarily halve the complete latency from input action is performed to result is visible on-screen.

It will be shown later that these assumptions are not hard requirements. So with those assumptions in mind, here's a diagram of the decisions involved when player A, controlling client A, presses the fire button:


Client A
Client B
Client C
Frame 1
Fire pressed (schedule for frame 5)
20 ms
Frame 2
Permission from A to advance to frame 5
Permission from C to advance to frame 5
Fire event received by B
Frame 3
The earliest frame the event could have been applied to at client B
Permission from B to advance to frame 5
Frame 4
Fire event received by C
Frame 5
Client A will schedule the fire event for frame 5 since the estimated arrival at client C will not be in time for frames < 5.
The event will be scheduled for frame 5, due to client C is known to having higher latency from A
Client C will apply the fire event at frame 5

When player A, using client A presses the fire button, client A must figure out which emulation frame this event should be applied on. Client A knows the delay to clients B and C, and it knows that if sent immediately, the message will arrive in the middle of frame 4 at client C - which is too late to apply to if events are applied at the start of frames. Client B will receive the event during frame 2. So, client A decides that frame 5 is the target frame for that event, and sends a message [event=fire,frame=5] to both clients B and C.

In order for something like this to work, the others clients must know when it is safe to start emulating frame 5 (because an input event scheduled for frame 5 cannot arrive after frame 5 emulation have started - unless emulation rewinding is implemented (it isn't) and works perfectly).

So, all clients must therefore send a message to all the other clients when it will no longer schedule events from frame 5. When this message is sent from a client, that client can only schedule input events for frame 6 or later. When the time comes to start emulating frame 5, each client will check that it has got that an advance-to-frame-5 message from each of the other clients. If not, the client will wait until all such acknowledgements have arrived.

Please note that the communication in the diagram is somewhat abstract. There isn't any advantage for a client to receive an input event for frame 5 before the frame 5 advance message, so the input events scheduled for frame 5 could very well be queued up and sent together with the frame 5 advance message. That could also contribute to less chances of having to retransmit network packets.

Also note that in the diagram, the frame advance message is sent just in time to arrive for the frame emulation for the "slowest client". In a practical system, one would add some delay to the calculation so the message is received a bit earlier. But it also advantageous to delay the frame-advance message as long as possible, to get as many input events as possible scheduled for that frame (or earlier).

In Reality...

In reality, one cannot know exactly how much time it will take to send a message to the other clients. Transfer times can fluctuate, and packet loss along the way could temporarily increase the latency quite a bit.

So instead, the clients must measure transfer time between themselves at all times, for example by sending "ping messages" to each other. The round-trip-time / 2 is probably a fair enough estimation for the one-way trip time between a client and another (in theory, there could be different latencies depending on which way the traffic flows).

The clients should ideally also store some history of transfer times between themselves, so they can take into account parameters such as recent average transfer time, and if the transfer time fluctuates, use a conservative estimate to reduce the chances of other clients stalling because of late messages.

Users could also be allowed to add a configurable delay to the net play game to further increase smoothness if the latencies fluctuate too much. The system could also in theory dynamically calculate and add a delay as needed based on the frequency of occurred pauses...

Synchronized Clocks

Synchronized clocks / common agreement on time may not be necessary for this netplay system (It is not directly required by the netplay protocol). It is also possible that allowing the peers to agree about the best synchronized time to emulate a certain frame could increase smoothness. After all, the decision on what frame to schedule an event for is based on knowing when the other clients want to start emulating it. But since peers throttle each other anyway, it may turn out to just work smoothly enough without.

There are several alternatives for synchronized clocks, if needed:

- Require that all players have accurate clocks on their computers, set by NTP (network time protocol.
- Have a built-in NTP client and calculate offset to -and correct - local time as interpreted by the client.
- Use a third-party server to calculate local time offset.
- Use some kind of other distributed time protocol to agree on a common time (or at least calculate time differences).
- Have clients broadcast the time they rendered a given frame, and use latency information to calculate the corresponding local time.

Input Event Order

It isn't enough to ensure that all clients get an advance-to-frame-X message after having received all input events scheduled for frame X. A game will only run without desync if all input events for a frame are applied in the same order within that frame on all clients.

This can be achieved in several ways, here is one:

- Have each client associate a random number with each input event generated.
- All clients will order the input events based on the associated (randomly chosen) number.
- In case of two input events with the same (randomly chosen) number, sort the list further based on client IDs. We can require that all clients generate a unique ID for itself which is known by the other clients.

The end result of this should be that all clients will agree on the order of input events, and all clients will also have the complete set of events scheduled for frame X, since they all waited for an advance-to-frame-X message from the other clients.

Partial Frame Updates

Instead of synchronizing input events on a per-frame basis, one could split the Amiga frame into more parts, and synchronize each part individually. For example, one could synchronize input events on “half-frames” instead. This means that there would be 10 ms between each half-frame advance message instead of 20 ms between each frame-advance message.

This could lower latency a bit, because if an input event is “just too late” to be scheduled for frame X, instead of having to schedule it for frame X + 1, it could be scheduled for frame X + 0.5 instead. On the flip side, it will cause more packets to be sent which could increase the chance of a lost packet / retransmit which would introduce a short stall…

About ENet Channels and Reliably Delivery

Reliable in-order delivery is a requirement for the netplay system to work properly. Reliable in-order delivery means that a packet lost will temporarily stall the communication.

To reduce the potential impact of this, only the netplay state data which must be reliable and in-order should be sent in the “same channel”. Other messages, such as latency measurements, chat messages, frame acknowledgements (with state checksums) and statistics should be sent through another channel, so retransmitted packets for these purposes do not affect the emulation smoothness.

ENet conveniently supports virtual channels on top of UDP, with in-order and reliably delivery being per-channel.

Comparison to the Client-Server Model

If the event scheduling is “aggressive”, that is, events are scheduled as early as possible (and frame advance messages are sent as late as possible), one would likely get more jitter than with the old FS-UAE net play method. This can be mitigated with good predictions, and not least by a adding a certain “slack” in the predictions (be a bit pessimistic).

While the peer-to-peer system is more fair in that a player cannot run a local server to get near zero latency for himself, the users can still have different input latencies. In the example, client B will have the lowest latency since he has the shortest “longest one-way time to each client”. With more information and statistics passed around to the peer, the system could in theory “penalize” client B a bit, so it will have the same input latency as A and C. In a two-player game, the input latencies for both players will be more or less identical.

Appendix A: Rollback Techniques / State Rewinding

Note: The netplay method described in the rest of this document does not use rollback-based techniques, so you can stop reading now, if you want to.

There are at least two areas where state rewinding could be used to combat network latencies and delays:

1. It could relax the requirement that clients must wait on an advance-to-frame-X message.
2. It could be used in combination with input predictions, aggressively assuming specific inputs from players, go with that without waiting for confirmation from other peers, and roll back if needed (if the prediction was wrong).

For the first point, the idea is that when the time comes, clients can go ahead and emulate frame X without knowing for certain that they have received all events for frame X. With good latency monitoring and prediction, the client most likely has received the input events already. If it turns out an input event is received after frame emulation has started, the state must be rewound and the frame emulated again including the new input event(s).

But the advantage here isn't that great. It does not really reduce latency itself, but it can prevent clients from stalling due to missing network packets, but it really only makes a positive difference when only the frame-advance message itself is delayed.

Instead, let us look at the second point, which is more interesting.

Applying Local Input Earlier (or Immediately)

The real advantage of using rollback techniques is to allow applying an input event before it would otherwise be safe. I.e., if player A presses fire, client A can choose to apply the input event to the emulation immediately, without knowing if the message will reach the other clients in time to apply to the same frame (most likely it will not, that is much of the point of the feature).

The advantage is obvious. The player performing input will experience exactly the same input lag as he does with an offline game. The consequence is that the other clients must be prepared to re-emulate the last few frames when it turns out that an input event arrives too late for an already emulated frame.

An example should help to clarify what happens (we use the input event from the previous diagram and with the same clients as an example):

- Frame 1 is emulated (nothing exciting happens here in this example)
- Player A presses fire - input event message is sent to other clients.
- Frame 2:
    - Frame 2 is emulated by client A, and client A applies the fire event.
    - Frame 2 is also emulated for client B and C, but they have no knowledge of the input event from A yet, so in this “reality”, the fire event has not occurred for them, and client B and C’s frame 2 are different from client A’s frame 2.
- Frame 3:
    - Frame 3 is emulated by client A as normal.
    - Now the input event (destined for frame 2) has been received by client B, and it knows client A applied it on frame 2. Therefore it must undo frame 2, re-emulated it with the fire event, and then it can emulate frame 3.
    - Client C has not seen the fire event yet, and emulates frame 3 according to client C’s reality.
    - Now A and B have equal frames for frame 3, but client C differs.
- Frame 4:
    - Nothing exciting happens here, all clients emulate frame 4.
    - A and B will still have equal frames for frame 4, but client C will have a different (“wrong)” frame 4.
- Frame 5:
    - By now, the fire event has reached client C, and it sees that the fire event should have been applied to frame 2. Client C must then rewind frame 4, 3, and 2, and re-emulate frame 2, 3 and 4 before it can emulate frame 5.
    - Assuming no other input events have occurred in the meantime, all clients now agree on frame 5.

Using the game Pong as an example also highlights a case where this optimistic technique really helps. **With traditional netplay code**, there will be some lag between the user performs an input, and the input is applied in the emulation. Thus, there will be some edge cases where the player would have hit the ball with the paddle, but due to input latency, the movement is “too late”, and the paddle just missed the ball.

With rollback-based netplay code the following would happen instead. Lets use an example where the two players have a network latency between them corresponding to three Amiga frames. When the ball nears the paddle for player A, and the player moves the paddle, there is no (network-caused) input lag before the paddle is moved, and the paddle just barely hits the ball successfully instead of missing it.

What would the other player see? Player B would actually see Player A miss the ball, since the input events causing the paddle to hit the ball isn’t known yet. Then, 3 frames later, when the input event has arrives, client B will re-emulated the last 3 frames before emulating the next frame, and suddenly, instead of the ball having left the playfield, it is now partly on the way back to Player B, and Player A's paddle has suddenly moved into position behind the ball.

So to summarize this specific Pong example:

- Player A will be able to more easily hit the ball since there is no network-related input lag.
- Player B will see something looking like time travel, where player A jumps back in time in order to prevent himself from losing. The Amiga display will "warp" to display the new reality, the more frames have occurred in the meantime, the bigger the visible changes.

In addition to this, Player B will also experience "problems" with audio. He will not hear the initial parts of the "clank" sample, and the "clank" will suddenly start playing. In addition, depending on the implementation of audio buffering, he can also hear discontinuities in audio/music playback and/or require higher latency in audio to prevent this.

One way to summarize the differences between the traditional approach and using rollbacks is: **You trade input latency with delay until other players see a player's actions.** For Pong, this may not matter that much, as it will take some time until the ball gets near the other player's paddle anyway. In other cases/games, the tradeoff may not be so advantageous.

Pros:

- A player can respond to events with almost no input lag (this is especially valid for events which will not be rolled back).

Cons:

- Other players will only see an action performed by a player after a delay, and they will not see the first bit of the animation for the action.
- Other players will not hear the first part of audio samples played as result of a player's actions.
- When a player performs input, the graphics for other players will jump and skip a bit. In a scrolling game, the entire screen may jump around a bit, in other cases, only some sprites may warp around and/or some visible actions will be "mysteriously" undone.
- Audio may have to be played with larger audio buffers (more audio latency), or instead audio discontinuities may be heard.

There are unfortunately a couple of problems with using rollback techniques in FS-UAE (in addition to the inherent smoothness issues):

- A successful system also would have to be able to rewind and replay up to several Amiga frames in a short time span (when predictions fail). This could be prohibitively expensive (CPU-wise) for many computers (emulating for example 4 Amiga frames in

cycle-accurate mode in the time span for a single frame is a very heavy operation).
- The state rewind mechanism currently in place in UAE might not be accurate enough for netplay purposes - a rewound / replayed state may not end up identical to the first state, which can cause desyncs. State saving/loading would probably have to go through a thorough review.
- The current state loading function in UAE seems slow and may result in even less time available for re-emulating frames.

Of course, the smoothness and performance issues can be reduced somewhat by instead of applying input events immediately, one can choose to apply them a bit later (but still earlier than would have happened with traditional netplay code). In this case, there will be fewer frames to re-emulate, less dramatic time warps, but also with some input latency.