

Deterministic Lockstep

Keeping simulations in sync by sending only inputs

Posted by Glenn Fiedler (<http://web.archive.org/web/20181107181426/https://gafferongames.com/about>) on Saturday, November 29, 2014

Introduction

Hi, I'm [Glenn Fiedler](http://web.archive.org/web/20181107181426/https://gafferongames.com/about) (<http://web.archive.org/web/20181107181426/https://gafferongames.com/about>) and welcome to [Networked Physics](http://web.archive.org/web/20181107181426/https://gafferongames.com/categories/networked-physics/) (<http://web.archive.org/web/20181107181426/https://gafferongames.com/categories/networked-physics/>).

In the [previous article](http://web.archive.org/web/20181107181426/https://gafferongames.com/post/introduction_to_networked_ph) (http://web.archive.org/web/20181107181426/https://gafferongames.com/post/introduction_to_networked_ph) we explored the physics simulation we're going to network in this article series. In this article specifically, we're going to network this physics simulation using **deterministic lockstep**.

Deterministic lockstep is a method of networking a system from one computer to another by sending only the *inputs* that control that system, rather than the *state* of that system. In the context of networking a physics simulation, this means we send across a small amount of input, while avoiding sending state like position, orientation, linear velocity and angular velocity per-object.

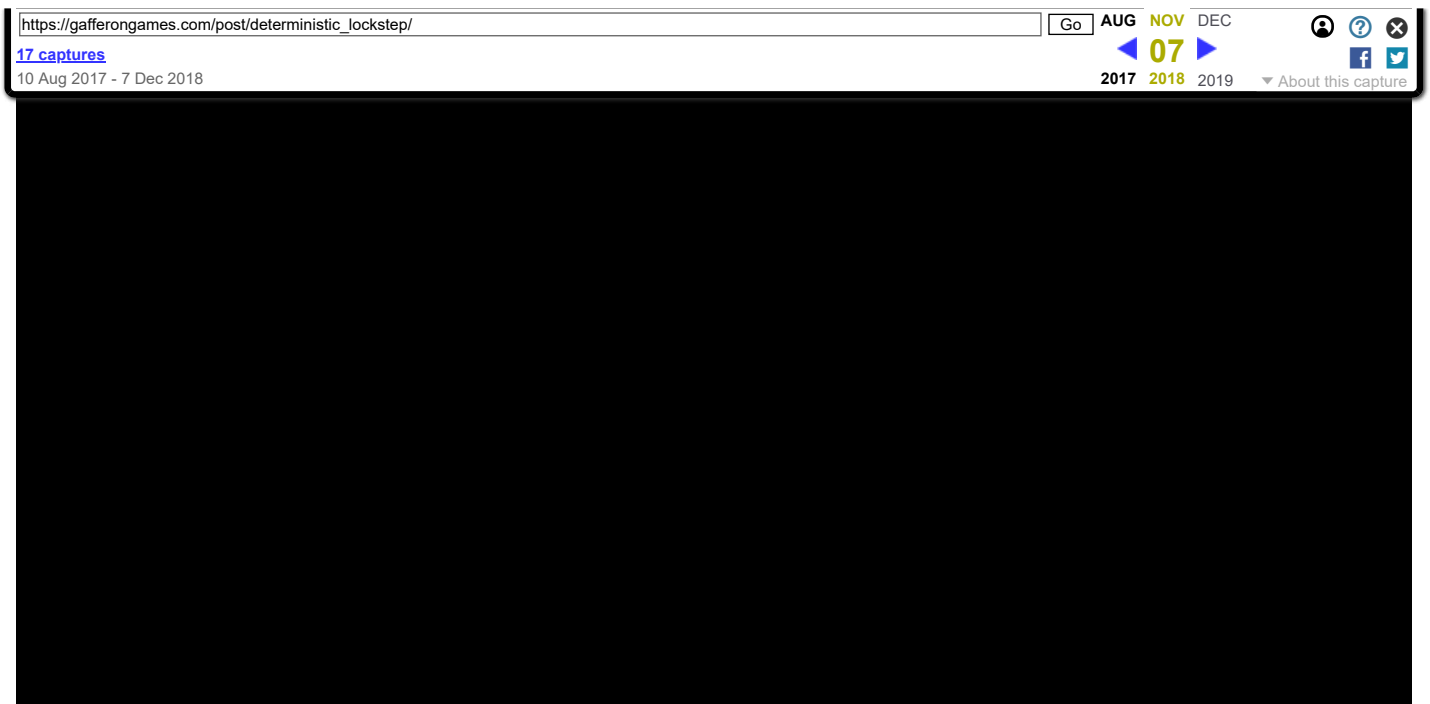
The benefit is that bandwidth is proportional to the size of the input, not the number of objects in the simulation. Yes, with deterministic lockstep you can network a physics simulation of one million objects with the same bandwidth as just one.

While this sounds great in theory, in practice it's difficult to implement deterministic lockstep because most physics simulations are not deterministic. Differences in floating point behavior between compilers, OS's and even instruction sets make it almost impossible to guarantee determinism for floating point calculations.

Determinism

Determinism means that given the same initial condition and the same set of inputs your simulation gives exactly the same result. And I do mean *exactly* the same result.

Not close. Not near enough. **Exactly the same**. Exact down to the bit-level. So exact, you could take a checksum of your entire physics state at the end of each frame and it would be identical.



Above you can see a simulation that is *almost* deterministic. The simulation on the left is controlled by the player. The simulation on the right has exactly the same inputs applied with a two second delay starting from the same initial condition. Both simulations step forward with the same delta time (a necessary precondition to ensure exactly the same result) and both simulations apply the same inputs. Notice how after the smallest divergence the simulation gets further and further out of sync. This simulation is **non-deterministic**.

What's going on is that the physics engine I'm using ([Open Dynamics Engine](http://web.archive.org/web/20181107181426/http://www.ode.org/) (<http://web.archive.org/web/20181107181426/http://www.ode.org/>)) uses a random number generator inside its solver to randomize the order of constraint processing to improve stability. It's open source. Take a look and see! Unfortunately this breaks determinism because the simulation on the left processes constraints in a different order to the simulation on the right, leading to slightly different results.

Luckily all that is required to make ODE deterministic on the same machine, with the same compiled binary and on the same OS (is that enough qualifications?) is to set its internal random seed to the current frame number before running the simulation via `dSetRandomSeed`. Once this is done ODE gives exactly the same result and the left and right simulations stay in sync.



And now a word of warning. Even though the simulation above is deterministic on the same machine, that does not necessarily mean it would also be deterministic across different compilers, a different OS or different machine architectures (eg. PowerPC vs. Intel). In fact, it's probably not even deterministic between debug and release builds due to floating point optimizations.

Floating point determinism is a complicated subject and there's no silver bullet.

For more information please refer to this [article](#)

(http://web.archive.org/web/20181107181426/https://gafferongames.com/post/floating_point_determinism/).

Networking Inputs

Now let's get down to implementation.

Our example physics simulation is driven by keyboard input: arrow keys apply forces to make the player cube move, holding space lifts the cube up and blows other cubes around, and holding 'z' enables katamari mode.

How can we network these inputs? Must we send the entire state of the keyboard? No. It's not necessary to send the entire keyboard state, only the state of the keys that affect the simulation. What about key press and release events then? No. This is also not a good strategy. We need to ensure that exactly the same input is applied on the right side, at exactly the same time, so we can't just send 'key pressed', and 'key released' events over TCP.

What we do instead is represent the input with a struct and at the beginning of each simulation frame on the left side, sample this struct from the keyboard:

```
struct Input
{
    bool left;
    bool right;
    bool up;
    bool down;
    bool space;
    bool z;
};
```

https://gafferongames.com/post/deterministic_lockstep/ Go

AUG NOV DEC

17 captures

07

10 Aug 2017 - 7 Dec 2018

2017 2018 2019

About this capture

And here's the key part: the simulation on the right can only simulate frame n when it has the input for that frame. If it doesn't have the input, it has to wait.

For example, if you were sending across using TCP you could simply send the inputs and nothing else, and on the other side you could read the packets coming in, and each input received corresponds to one frame for the simulation to step forward. If no input arrives for a given render frame, the right side can't advance forward, it has to wait for the next input to arrive.

So let's move forward with TCP, you've disabled Nagle's Algorithm (http://web.archive.org/web/20181107181426/https://en.wikipedia.org/wiki/Nagle's_algorithm), and you're sending inputs from the left to the right simulation once per-frame (60 times per-second).

Here it gets a little complicated. Since we can't simulate forward unless we have the input for the next frame, it's not enough to just take whatever inputs arrive over the network and then run the simulation on inputs as they arrive because the result would be very jittery. Data sent across the network at 60HZ doesn't typically arrive nicely spaced, $1/60$ th of a second between each packet.

If you want this sort of behavior, you have to implement it yourself.

Playout Delay Buffer

Such a device is called a playout delay buffer.

Unfortunately, the subject of playout delay buffers is a patent minefield. I would not advise searching for "playout delay buffer" or "adaptive playout delay" while at work. But in short, what you want to do is buffer packets for a short amount of time so they *appear* to be arriving at a steady rate even though in reality they arrive somewhat jittered.

What you're doing here is similar to what Netflix does when you stream a video. You pause a little bit initially so you have a buffer in case some packets arrive late and then once the delay has elapsed video frames are presented spaced the correct time apart. If your buffer isn't large enough then the video playback will be hitchy. With deterministic lockstep your simulation behaves exactly the same way: showing hitches when the buffer isn't large enough to smooth out the jitter. Of course, the cost of increasing the buffer size is additional latency, so you can't just buffer your way out of all problems. At some point the user says enough! That's too much latency added. No sir, I will not play your game with 1 second of extra delay :)

My playout delay buffer implementation is really simple. You add inputs to it indexed by frame, and when the very first input is received, it stores the current local time on the receiver machine and from that point on delivers packets assuming they should play at that time + 100ms. You'll likely need to something more complex for a real world situation, perhaps something that handles clock drift, and detecting when the simulation should slightly speed up or slow down to maintain a nice amount of buffering safety (being "adaptive") while minimizing overall latency, but this is reasonably complicated and probably worth an article in itself.

The goal is that under average conditions the playout delay buffer provides a steady stream of inputs for frame n , $n+1$, $n+2$ and so on, nicely spaced $1/60$ th of a second apart with no drama. In the worst case the time arrives for frame n and the input hasn't arrived yet it returns null and the simulation is forced to wait. If packets get bunched up and delivered late, it's possible to have multiple inputs ready to dequeue per-frame. In this case I limit to 4 simulated frames per-render frame so the simulation has a chance to catch up, but doesn't simulate for so long that it falls further behind, aka. the "spiral of death".

https://gafferongames.com/post/deterministic_lockstep/ Go

AUG NOV DEC

17 captures

07

10 Aug 2017 - 7 Dec 2018

2017 2018 2019

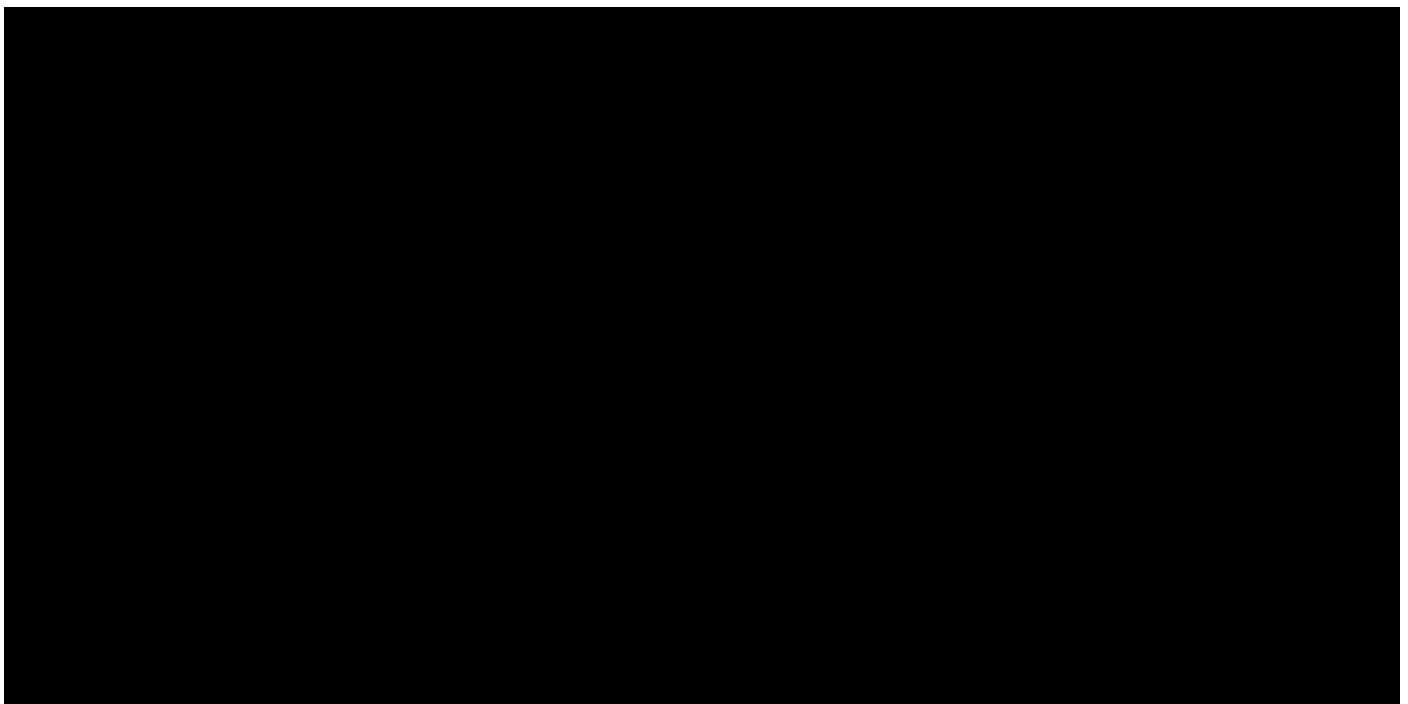
About this capture

Using this playout buffer strategy and sending inputs across TCP we ensure that all inputs arrive reliably and in-order. This is convenient, and after all, TCP is designed for exactly this situation: reliable-ordered data.

In fact, It's a common thing out there on the Internet for pundits to say stuff like:

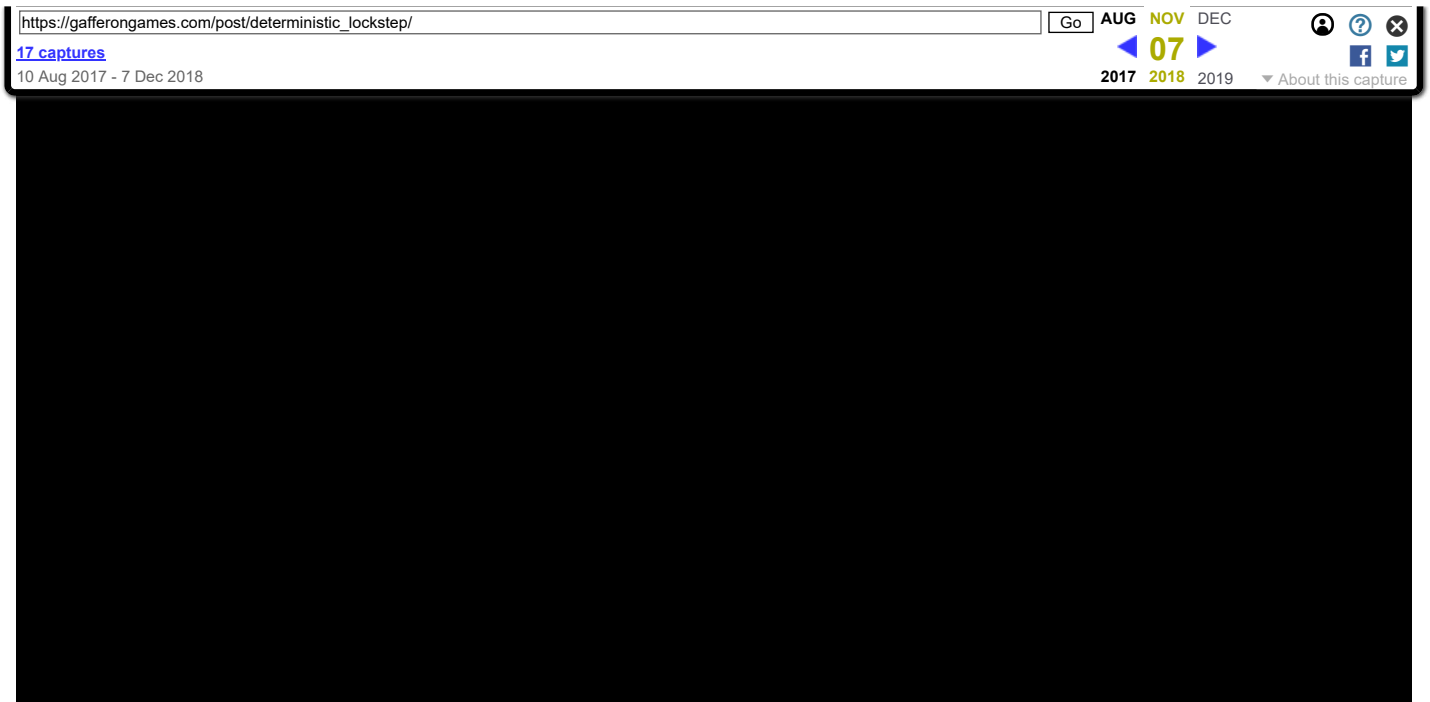
- If you need reliable-ordered, you can't do better than TCP!
(http://web.archive.org/web/20181107181426/https://www.reddit.com/r/gamedev/comments/1tvbe0/is_it)
- Your game doesn't need UDP (yet)
(<http://web.archive.org/web/20181107181426/https://thoughtstreams.io/glyph/your-game-doesnt-need-udp-yet/>)

But I'm here to tell you this kind of thinking is **dead wrong**.



Above you can see the simulation networked using deterministic lockstep over TCP at 100ms latency and 1% packet loss. If you look closely on the right side you can see hitches every few seconds. What's happening here is that each time a packet is lost, TCP has to wait $RTT \times 2$ while it is resent (actually it can be much worse, but I'm being generous...). The hitches happen because with deterministic lockstep the right simulation can't simulate frame n without input n , so it has to pause to wait for input n to be resent!

That's not all. It gets significantly worse as latency and packet loss increase. Here is the same simulation networked using deterministic lockstep over TCP at 250ms latency and 5% packet loss:



Now I will concede that if you have no packet loss and/or a very small amount of latency then you very well may get acceptable results with TCP. But please be aware that if you use TCP it behaves *terribly* under bad network conditions.

Can we do better than TCP?

Can we beat TCP at its own game. Reliable-ordered delivery?

The answer is an emphatic **YES**. But *only* if we change the rules of the game.

Here's the trick. We need to ensure that all inputs arrive reliably and in order. But if we send inputs in UDP packets, some of those packets will be lost. What if, instead of detecting packet loss after the fact and resending lost packets, we redundantly include *all inputs* in each UDP packet until we know for sure the other side has received them?

Inputs are very small (6 bits). Let's say we're sending 60 inputs per-second (60fps simulation) and round trip time we know is going to be somewhere in 30-250ms range. Let's say just for fun that it could be up to 2 seconds worst case and at this point we'll time out the connection (screw that guy). This means that on average we only need to include between 2-15 frames of input and worst case we'll need 120 inputs. Worst case is $120 \times 6 = 720$ bits. That's only 90 bytes of input! That's totally reasonable.

We can do even better. It's not common for inputs to change every frame. What if when we send our packet instead we start with the sequence number of the most recent input, and the 6 bits of the first (oldest) input, and the number of un-acked inputs. Then as we iterate across these inputs to write them to the packet we can write a single bit (1) if the next input is different to the previous, and (0) if the input is the same. So if the input is different from the previous frame we write 7 bits (rare). If the input is identical we write just one (common). Where inputs change infrequently this is a big win and in the worst case this really isn't that bad. 120 bits of extra data sent. Just 15 bytes overhead worst case.

Of course another packet is required from the right simulation to the left so the left side knows which inputs have been received. Each frame the right simulation reads input packets from the network before adding them to the playout delay buffer and keeps track of the most recent input it has received and sends this back to the left as an "ack" or acknowledgment for inputs.

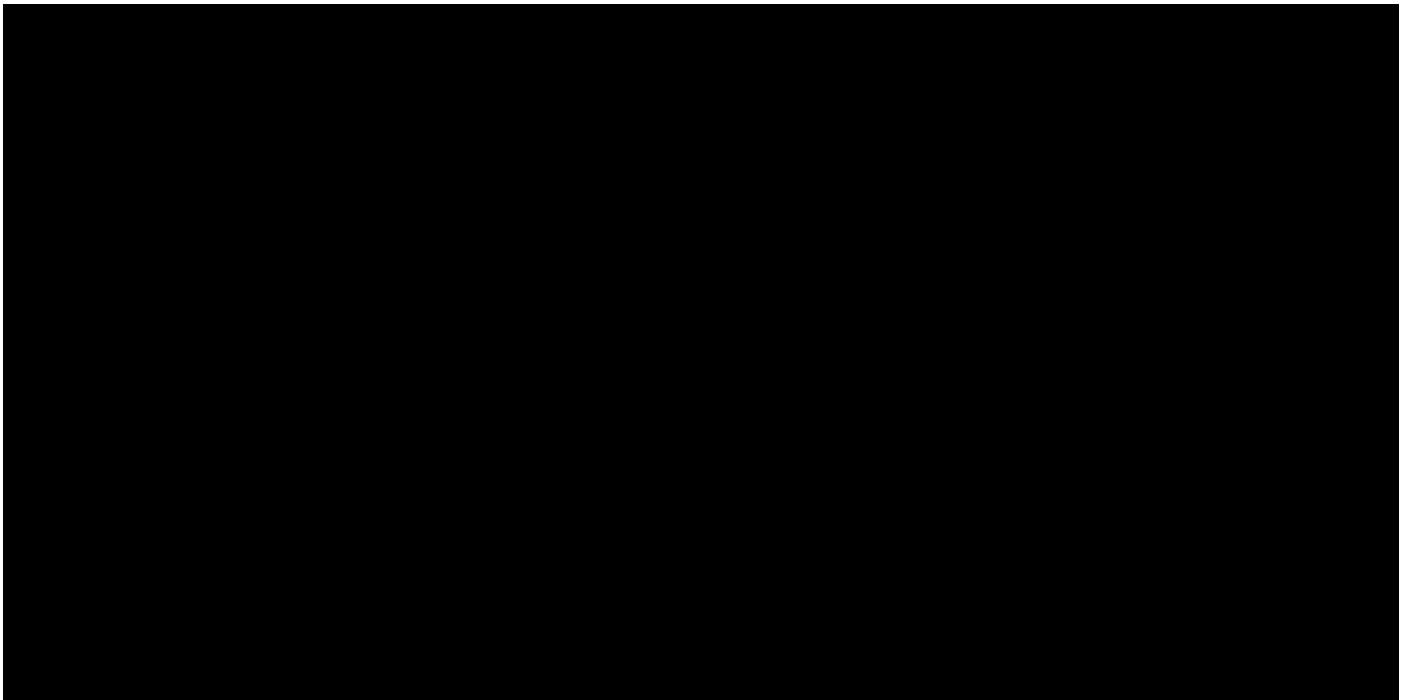
Flawless Victory

We have beaten TCP by changing the rules of the game.

Instead of “implementing 95% of TCP on top of UDP” we have implemented something *totally different* and better suited to our requirements. A protocol that redundantly sends inputs because we know they are small, so we never have to wait for retransmission.

So exactly how much better is this approach than sending inputs over TCP?

Let's take a look...





The video above shows deterministic lockstep synchronized over UDP using this technique with **2 seconds** of latency and **25% packet loss**. Imagine how awful TCP would look under these conditions.


So in conclusion, even where TCP should have the most advantage, in the only networking model that relies on reliable-ordered data, we can still easily whip its ass with a simple protocol built on top of UDP.

NEXT ARTICLE: [Snapshot Interpolation](http://web.archive.org/web/20181107181426/https://gafferongames.com/post/snapshot_interpolation/)

(http://web.archive.org/web/20181107181426/https://gafferongames.com/post/snapshot_interpolation/)

 (<http://web.archive.org/web/20181107181426/https://www.linkedin.com/in/glennfiedler/>)

 (<http://web.archive.org/web/20181107181426/https://twitter.com/gafferongames>)

 (<http://web.archive.org/web/20181107181426/https://github.com/gafferongames>)

https://gafferongames.com/post/deterministic_lockstep/

Go

AUG

NOV

DEC

◀

07

▶

2017

2018

2019

About this capture