Lock-Step Simulation Is Child's Play (Experience Report)

JOACHIM BREITNER, University of Pennsylvania, USA
CHRIS SMITH, Google, USA

Implementing multi-player networked games by broadcasting the player's input and letting each client calculate the game state – a scheme known as *lock-step simulation* – is an established technique. However, ensuring that every client in this scheme obtains a consistent state is infamously hard and in general requires great discipline from the game programmer. The thesis of this report is that in the realm of functional programming – in particular with Haskell's purity and static pointers – this hard problem becomes almost trivially easy.

We support this thesis by implementing lock-step simulation under very adverse conditions. We extended the educational programming environment CodeWorld, which is used to teach math and programming to middle school students, with the ability to create and run interactive, networked multi-user games. Despite providing a very abstract and high-level interface, and without requiring any discipline from the programmer, we can provide consistent lock-step simulation with client prediction.

3

CCS Concepts: • **Software and its engineering → Functional languages**; • **Computer systems organization → *Peer-to-peer architectures***;

Additional Key Words and Phrases: teaching, lock-step simulation, distributed computation

ACM Reference Format:
Joachim Breitner and Chris Smith. 2017. Lock-Step Simulation Is Child's Play (Experience Report). *Proc. ACM Program. Lang.* 1, ICFP, Article 3 (September 2017), 15 pages.
https://doi.org/10.1145/3110247

1 INTRODUCTION

Networked multi-user games must tackle the challenge of ensuring that all participating players on a network with potentially significant latency still see the same game state. In some circumstances, an appealing choice is *lock-step simulation*. In this scheme, which dates back to the age of Doom, the state of the game itself is never transmitted over the network. Instead, the clients exchange information about their player's interactions – as abstract game moves or just the actual user input events – and each client independently calculates the state of the game.

Of course, this only works as intended if all clients end up with the same state. The technique is fraught with danger if the programmer is not very careful and disciplined about managing that state. Terrano and Bettner [2001], who implemented the network code for the real time strategy games Age of Empires 1 & 2, report:

> As much as we check-summed the world, the objects, the pathfinding, targeting and every other system – it seemed that there was always one more thing that slipped just under the radar. [...] Part of the difficulty was conceptual – programmers were not

2475-1421/2017/9-ART3
https://doi.org/10.1145/3110247

used to having to write code that used the same number of calls to random within the simulation.

More drastic words were voiced by Smith [2011], also a video game software engineer:

One of the most vile bugs in the universe is the desync bug. They're mean sons of bitches. The grand assumption to the entire engine architecture is all players being fully synchronous. What happens if they aren't? What if the simulations diverge? Chaos. Anger. Suffering.

The pitfalls facing a programmer implementing lock-step simulation include reading the system clock, querying the random number generator, other I/O, uninitialized memory, and local or hidden statefulness. In short: side effects! What if we chose a programming language without such side-effects? Would these problems disappear? Intuitively, we expect that **pure functional programming makes lock-step simulation easy**.

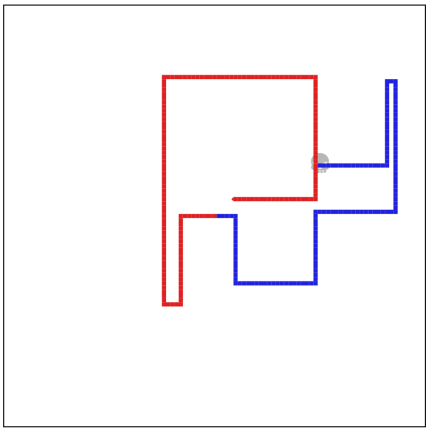
Fig. 1. The Snake game

This experience report corroborates our expectation. We have implemented lock-step simulation in Haskell under very adverse conditions. The authors of the quotes above are professional programmers working on notable games. They can be expected to maintain a certain level of programming discipline, and to tolerate additional complexity. Our implementation is part of CodeWorld[1], an educational, web-based programming environment used to teach mathematics and coding to students as early as middle school. These children, who are just learning to code, can write and run arbitrary game logic, using a simple API, without adhering to any additional requirements or coding discipline. Nevertheless, we still guarantee consistent lock-step simulation and avoid the dreaded desync bug.

The main contributions of this experience report are:

- With a bold disregard for pesky implementation detail, we design a natural extension to CodeWorld's existing interfaces that can describe multi-user interactive programs in as straightforward, simple and functional a manner as possible (Section 3.1).
- We identify a complication – unwanted capture of free variables – which can thwart consistency of such a program. We solve it using either using the module system (Section 3.2) or the Haskell language extension *static pointers* (Section 3.3).
- We explain how to implement this interface. Despite its abstractness, we present an eventually consistent implementation that works for arbitrary client code. Our implementation includes *client prediction* to react immediately to local input while still reconciling delayed input from other users (Section 4).
- Haskell's promise of purity is muddied by underspecification of floating point transcendental functions. Replacing these with deterministic approximations recovers the consistency that we rely upon (Section 5.1).
- We show that, even with no knowledge of the structure of the program's state, our approach still allows us to smooth out artifacts that arise due to network latency (Section 5.2).
- Overall, we show that pure functional programming makes lock-step simulation easy.

[1] https://code.world/haskell

2 CODEWORLD

In this section, we give a brief overview of how students interact with the CodeWorld environment, the programming interfaces that are provided by CodeWorld and how student programs are executed. Many of the figures illustrating this paper are created by students. These and more can be found in the CodeWorld gallery at https://code.world/gallery.html.

To ease deployment, students need only a web browser to use CodeWorld. They write their code with an integrated editor inside the browser. Programs are written in Haskell, which the CodeWorld server compiles to JavaScript using GHCJS [Stegeman and Mackenzie 2017] and sends that back to browser to execute in a canvas beside the editor. These programs are always graphical: students create static pictures, then animations, and finally interactive games and other activities.

2.1 Two Flavors of Haskell

The standard Haskell language is not an ideal vessel for the children in CodeWorld's target audience. Therefore, CodeWorld by default provides a specially tailored educational environment. In this mode, a custom prelude is used to help students avoid common obstacles. Graphics primitives are available without an import, to create appealing visual programs. Functions of multiple arguments are not curried but rather take their arguments in a tuple, both to improve error messages and match mathematical notation that students are already learning. Finally, a single Number type (isomorphic to Double) is provided to avoid the need for type classes, and the RebindableSyntax language extension makes literals monomorphic. Compiler error messages are post-processed to make them more intelligible to the students. Nevertheless, the code students write is still Haskell, and is accepted by GHC.

However, at https://code.world/haskell instead of https://code.world/, one finds a standard Haskell environment, with full access to the standard library. In this paper we focus on the latter variant.

2.2 API Design Principles

An important principle of CodeWorld is to provide students with the simplest possible abstraction for a given task. This allows them to concentrate on the ideas they want to express and think clearly about the meaning of their code, and hides as many low-level details as possible.

The first and simplest task that students face is to produce a static *drawing*. This is done with the abstract data type Picture, with a simple compositional API (Figure 3) which was heavily inspired by the Gloss library [Lippmeier 2017]. Complex pictures are built by combining and transforming simple geometric objects. The entry point used for this has the very simple type

```
drawingOf :: Picture → IO ()
```

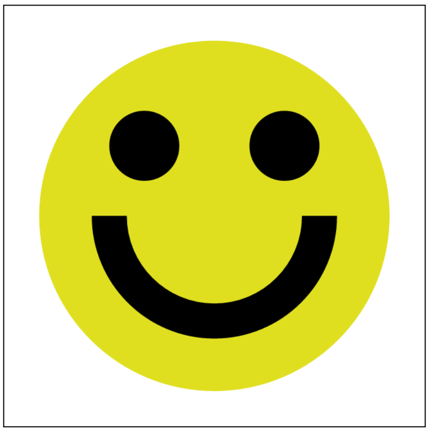

Fig. 2. Smiley

This function takes care of the details of displaying the student's picture on the screen, redrawing upon window size changes and so on. So all it takes for a student to get the computer to smile like in Figure 2 is to write

```
import CodeWorld

smiley = translated (−4) 4 (solidCircle 2) & translated 4 4 (solidCircle 2) &
         thickArc 2 (−pi) 0 6 & colored yellow (solidCircle 10)

main = drawingOf smiley
```

```
data Picture -- abstract

-- Various geometric shapes (circle, rectangle, arc, polygon etc.) are
-- available, and can either be filled or equipped with a thickness. E.g.:

-- Parameter: radius
solidCircle :: Double → Picture
-- Parameters: thickness, radius, start angle and end angle
thickArc    :: Double → Double → Double → Double → Picture

-- Pictures can be transformed and overlaid
colored     :: Color →                  (Picture → Picture)
translated  :: Double → Double →        (Picture → Picture)
rotated     :: Double →                 (Picture → Picture)
scaled      :: Double → Double →        (Picture → Picture)
(&)         :: Picture → Picture → Picture
```

Fig. 3. An excerpt of CodeWorld's Picture API

As a next step, the students can create *animations* and *simulations* to make their pictures move, before eventually making their programs react to user input in *interactions*. The game in Figure 4 is a typical interaction, where the player saves flying Grandma from various obstacles by attaching balloons or parachutes to her wheelchair.

These are created by calling the following interface:

```
interactionOf :: world
              → (Double → world → world)
              → (Event → world → world)
              → (world → Picture)
              → IO ()
```

In a typical call

```
main = interactionOf start step handle draw
```

the student passes four arguments, namely:

(1) an initial state, start,
(2) a time step function, step, which calculates an updated state as time passes,
(3) an event handler function, handle, which calculates an updated state when the user interacts with the program and
(4) a visualization function, draw, to depict the current state as a Picture.

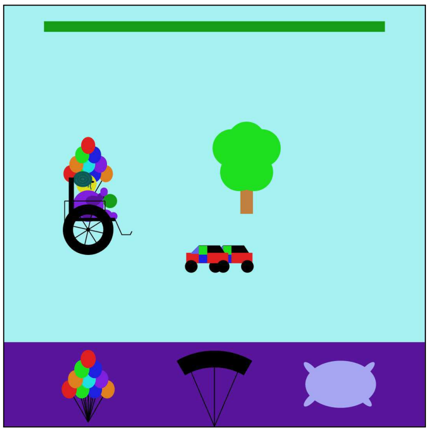
Fig. 4. Yo Grandma, © Sophia Hanna (6th grade)

The Event type, shown in Figure 5, is a simple algebraic data type that describes the press or release of a key or mouse button, or a movement of the mouse pointer.

The type of the state, world, is chosen by the user and consists of the domain-specific data needed by the program. The world type is completely unconstrained, and this will be an important factor influencing our design. It need not even be serializable, nor comparable for equality. In particular, the state may contain first-class functions and infinite lazy data structures. One way that students commonly make use of this capability is by defining infinite lazy lists of anticipated future events, based on a random number source fetched before the simulation begins.

```
type Point = (Double, Double)
data Event = KeyPress Text
           | KeyRelease Text
           | MousePress MouseButton Point
           | MouseRelease MouseButton Point
           | MouseMovement Point
data MouseButton = LeftButton | MiddleButton | RightButton
```

Fig. 5. The Event type

3 AN INTERFACE FOR MULTI-PLAYER GAMES

We would like students to extend their programming to networked multi-user programs, so that they can invite their friends to join over the internet and collaborate together on a drawing, fight each other in a fierce duel of Snake, or interact in any other way the student designs and implements. In this section, we turn our attention to choosing an API for such a task.

3.1 Wishful Thinking

Let us apply "API design by wishful thinking", and ask: What is the most convenient abstract model of a multi-player game we can hope for, independent of implementation concerns or constraints? As experienced programmers, our thoughts might drift to network protocols or message passing between independent program instances, each with its own local state. Our students, though, care about none of this, and ideally we would not burden them with it. In fact, motivated students have already implemented games to be played with classmates, using different keys on the same device. An example is shown in Figure 6, where the red player uses the keys W A S D and the blue player the keys ↑ ← ↓ →, in a race to consume more dots.

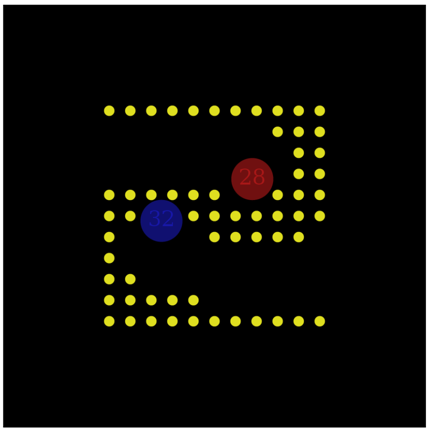

Fig. 6. Dot Grab, © Adrian Stark (7th grade)

Their games, which they have already designed, are described in terms of one shared global state. Why should the programming model change drastically simply because of one detail – that the code will now run on multiple nodes communicating over a network?

We conclude, then, that an interactive multi-user program is a generalization of an interactive single-user program, and the centerpiece of the API is still a single, global state, which is mutually acted upon by all players. Basing the API on interactionOf, we make only minimal changes to adapt to the new environment:

- A new first parameter specifies the number of players.
- The parameters start and step remain as they are.
- The handle parameter, though, ought to know *which* user pressed a certain button or moved their mouse, so it receives the player number (a simple Int) as an additional parameter.
- Different players may also see different views of the state, so the draw function also receives the player number for which it should render the screen – but it is free to ignore that parameter, of course.

All together, we arrive at the following "ideal" interface that we call *collaborations*, which allows students to build networked multi-player games and other activities:

```
collaborationOf :: Int
                → world
                → (Double → world → world)
                → (Int → Event → world → world)
                → (Int → world → Picture)
                → IO ()
```

A small example will clarify how this interface is used. The following code traces the mouse movements of two players using colored, fading circles, and Figure 7 shows this program in action. The green player is a bot that simply mirrors the red player's movements.

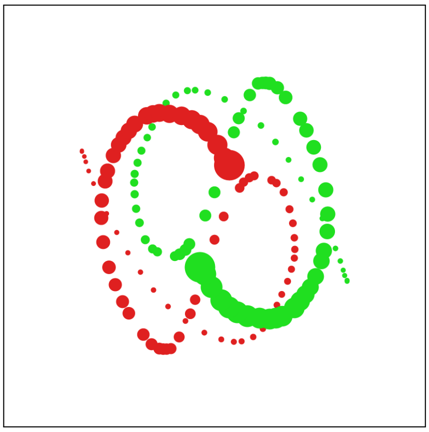
Fig. 7. Two players' mouse movements

```
import CodeWorld

type World = [(Color, Double, Double, Double)]

step :: Double → World → World
step dt dots = [(c, exp (−dt) ∗ r, x, y) | (c, r, x, y) ← dots, r ⩾ 0.1]

handle :: Int → Event → World → World
handle 0 (MouseMovement (x, y)) dots = (red,   1, x, y) : dots
handle 1 (MouseMovement (x, y)) dots = (green, 1, x, y) : dots
handle _ _                      dots = dots

draw :: Int → World → Picture
draw _ dots = mconcat [translated x y (colored c (solidCircle r)) | (c, r, x, y) ← dots]

main :: IO ()
main = collaborationOf 2 [ ] step handle draw
```

A collaboration begins with a lobby, featuring buttons to create or join a game. Upon creating a new game, the player is given a four-letter code to be shared with friends. Those friends may enter the four-letter code to join the game. Once enough players have joined, the game begins.

3.2 Solving Random Problems with the Module System

Like interactionOf before it, the parameters of collaborationOf provide enough information to completely determine the behavior of the program from the sequence of time steps and UI events that occur. Unlike interactionOf, however, a collaboration involves *more than one* use of the collaborationOf API, as the function is executed by each participating player. To ensure that there is a single, well-defined behavior, it is essential that all players run collaborationOf with the same arguments. Obviously, we need to ensure that all clients run the same program, and the CodeWorld server does so. But even with the same code, the arguments to collaborationOf can differ from client to client:

```
main = do
  r ← randomRIO (0, 1)
  collaborationOf numPlayers start step (handle r) draw
```

The event handling function now depends on I/O – specifically, the choice of a random number – and it is very unlikely that all clients happen to pick the same random number. Despite sharing the same code, the clients will disagree about the correct behavior of the system.

The problem is not the use of random numbers per se, but rather the unconstrained flow of client-specific state resulting from *any* I/O into the collaborationOf API via free variables in its parameters. Since most of the parameters to collaborationOf have function types, we cannot just compare them to establish consistency at runtime.

We solve this problem in two ways: one in the educational environment, and the other in the standard Haskell environment.

In the former, we have tight control over the set of library functions available to the student. No packages are exposed except for a custom standard library with a heavily customized Prelude module, and this library simply does not provide any functions to compose IO operations, such as as the monadic bind operators (≫=, ≫). This also rules out the use of Haskell's **do**-notation, which under the regime of RebindableSyntax requires an operator called (≫=) to be in scope. A valid Haskell program requires a top-level function main :: IO (), and since the only available way to obtain an IO () is through our API entry points (drawingOf, interactionOf, and so on), we know that all CodeWorld collaborations are of essentially the form main = collaborationOf . . . In particular, no I/O can be executed prior to the collaboration, and hence no client-dependent behavior is possible.

3.3 Solving Random Problems Syntactically

This solution is not suitable for the standard Haskell environment, where we do not want to restrict the user's access to the standard library. We can still prevent the user from using the results of client-specific I/O in arguments to collaborationOf. To accomplish this, we creatively use the work of Epstein et al. [2011], who sought to bring Erlang-like distributed computing to Haskell. They had to exchange functions over the network, which is possible by passing code references, as long as no potentially unserializable values are captured from the environment. To guarantee that, they introduced a Haskell language extension, *static pointers*, which introduces:

- a new type constructor StaticPtr a, which wraps values of type a,
- a new syntactic construct **static** foo, such that for any expression foo of type a, the expression **static** foo has type StaticPtr a, but is only valid if foo does not contain any locally bound free variables,
- a pure function deRefStaticPtr :: StaticPtr a → a, to unwrap the static pointer, and
- a pure function staticKey :: StaticPtr a → StaticKey which produces a key that – within one program – uniquely identifies a static pointer.

The requirement that StaticPtr values cannot have locally bound free variables turns out to be exactly what we need to prevent programs from smuggling client-specific state obtained with I/O actions into collaborations. We therefore further refine the API to require its arguments to be static pointers:

```
collaborationOf :: Int
                → StaticPtr world
                → StaticPtr (Double → world → world)
                → StaticPtr (Int → Event → world → world)
                → StaticPtr (Int → world → Picture)
                → IO ()
```

The mouse tracing program in Figure 7 must now change its definition of main to

```
main = collaborationOf 2 (static [ ]) (static step) (static handle) (static draw).
```

On the other hand, writing **static** (handle r) to smuggle in a randomly drawn number r, as in the example above, will fail at compile time. A somewhat more clever attempt, though, still causes problems:

```
main = do
   coinFlip ← randomIO
   let step = if coinFlip then static step1 else static step2
   collaborationOf 2 (static [ ]) step (static handle) (static draw)
```

This program is accepted by the compiler because the arguments to collaborationOf are indeed StaticPtr values of the right types, yet it raises the same questions when clients disagree on the choice of step function. While we cannot prevent this case at compile time, we can at least detect it at runtime. Static pointers can be serialized using the function staticKey :: StaticPtr a → StaticKey. Before a game starts, the participating clients compare the keys of their arguments to check that they match. This is a subtly different use of static pointers from the original intent of sending functions over a network in a message-passing protocol. We need not actually receive the original values on the remote end of our connections, but instead use the serialized keys only to check for consistency.

With this check in place – short of using unsafe features such as unsafePerformIO – we are confident that every client is indeed running the same functions. However, this forces our games to be entirely deterministic. This is a problem, since many games involve an element of chance! To restore the possibility of random behavior, we supply a random number source to use in building the initial state, with a consistent seed in all clients. The type of the start parameter is now StaticPtr (StdGen → world).

This completes our derivation of collaborationOf, which in its final form is

```
collaborationOf :: Int
                → StaticPtr (StdGen → world)
                → StaticPtr (Double → world → world)
                → StaticPtr (Int → Event → world → world)
                → StaticPtr (Int → world → Picture)
                → IO ()
```

4 FROM WISHFUL THINKING TO RUNNING CODE

How can we implement this interface? It turns out that our implementation options are severely narrowed down by the following requirements:

(1) We need to handle any code using the API. Given the educational setting of CodeWorld, we cannot require any particular discipline.
(2) The players need to see an eventually consistent state. They may temporarily have different ideas about the state of the world, but gain a shared understanding once everybody has received information about everybody's interactions.
(3) The effects of a player's own interactions are immediately visible to that player. Even a "local" interaction, such as selecting a piece in a game of Chess, will have to represented in the game state, and any latency here would make the user interface sluggish.

The first requirement in particular implies that the game state is completely opaque to us. This already rules out the usual client-server architecture, where only the central server manages the game state and the clients send abstract moves (e.g., "white moves the knight to e8") and render the game state that they receive from the server. We have neither insight into what constitutes an abstract move, nor how to serialize and transmit the game state.

We could avoid this problem by sending the raw UI Event instead of an abstract move to the server, and letting the server respond to each client with the Picture to show. This "dumb terminal" approach however would run afoul of our third requirement, as every user interaction would be delayed by the time it takes messages to travel to the server and back.

The requirement of immediate responsiveness implies that every client needs to manage its own copy of the game state, and being abstract in the game state implies that there is nothing else but the UI events that the clients can transmit to synchronize the state. In other words, lock-step simulation is the only way for us.

4.1 Types and Messages

We seek, then, to implement the API by exchanging UI events between clients. For the purposes of this paper, it does not matter how events are transmitted from client to client. The CodeWorld implementation uses a very simple relay server that broadcasts messages from one client to the others via WebSockets (a full-duplex server-client protocol for web applications), but peer-to-peer communication using WebRTC (a peer-to-peer protocol for web applications) or other methods would work equally well, as long as they deliver events reliably and in order.

Every such message obviously needs to contain the actual Event and the player number. In addition, it must contain a timestamp, so that each client applies the event at the same time despite differences in network latency. Otherwise – assuming a time-sensitive game with a non-trivial step function – the various clients would obtain different views of the world. Timestamps are Double values, measured in seconds since the start of the game.

```
type Timestamp = Double
type Player    = Int
type Message   = (Timestamp, Player, Event)
```

4.2 Resettable State

Having fixed the message type still leaves open the question of what to do with these messages, which is non-trivial due to the network latency.

Assume that 23.5 seconds into a real-time strategy game, I send my knights to attack the other player. My client sends the corresponding message (23.500, 0, MousePress LeftButton (20, 30)) to the other player. The message arrives, say, 100ms later. As mentioned before, the other player cannot simply let my knights set out a bit later. What else?

The classical solution [Terrano and Bettner 2001] is to not act on local events immediately, but add a delay of, say, 200ms. The message would be (23.700, 0, MousePress LeftButton (20, 30)), and assuming it reaches all other players in time, all are able to apply the event at precisely the same moment. This solution works well if the UI can somehow respond to the user's actions immediately, e.g. by letting the knight audibly confirm the command, so hide this delay from the user.

The luxury of such a separation is not available to us – according to the third requirement, each client must immediately apply its own events – and the message really has to have the timestamp 23.500. This leaves the other player, when it receives the message 100ms later, with no choice but to roll back the game state to time 23.500, apply my event, and replay the following 100ms. While rollback and replay are hard to implement in imperative programming paradigms, where every piece of data can have local mutable state, they are easy in Haskell, where we know that the value of type world really holds all relevant bits of the program's state.

One way of allowing such recalculation is to simply not store the state at all, and re-calculate it every time we draw the game screen. The function to do so would expect the game specification, the current time and the list of messages that we have seen so far, including the locally generated ones, and would calculate the game state. Its type signature would thus be

```
currentState :: Game world ⇒ Timestamp → [Message] → world
```

where the hypothetical type class Game captures the user-defined game logic; we introduce it here to avoid obscuring the following code listings by passing it explicitly around as an argument:

```
class Game world where
  start  :: world
  step   :: Double → world → world
  handle :: Player → Event → world → world
```

Assume, for a short while, that there was no step function, i.e. the game state changes only when there is an actual event. Then the timestamps are only required to put the events into the right order and to disregard events which are not yet to be applied (which can happen if the player's game time started at slightly different points in time):

```
currentState :: Game world ⇒ Timestamp → [Message] → world
currentState now messages = applyEvents to_apply start
  where to_apply = takeWhile (λ(t, _, _). t ⩽ now) (sortMessages messages)

sortMessages :: [Messages] → [Messages]
sortMessages = sortOn (λ(t, p, _). (t, p)) messages

applyEvents :: Game world ⇒ [Message] → world → world
applyEvents messages w = foldl apply w messages
  where apply w (_, p, e) = handle p e w
```

Eventually, every client receives the same list of messages, up to the interleaving of events from different players. After a stable sort by timestamp and player, the lists of events will be identical, so all clients will calculate the same game state.

4.3 A Few More Steps

This is nice and simple, but ignores the step function, which models the evolution of the state as time passes. Clearly, we have to call step before each event, and again at the end. In order to calculate the time passed since the last event, we also have to keep track of which timestamp a snapshot of the game state corresponds to:

```
currentState :: Game world ⇒ Timestamp → [Message] → world
currentState now messages = step (now − t) world
  where to_apply = takeWhile (λ(t, _, _). t ⩽ now) (sortMessages messages)
        (t, world) = applyEvents to_apply (0, start)

applyEvents :: Game world ⇒ [Message] → (Timestamp, world) → (Timestamp, world)
applyEvents messages ts = foldl apply ts messages
  where apply (t0, world) (t1, p, e) = (t1, handle p e (step (t1 − t0) world)))
```

Unfortunately, students would not be quite happy with this implementation. The step function is commonly used to calculate a single step in a physics simulation, which requires that it is called often enough to achieve a decent simulation frequency.

For instance, when simulating a projectile, a common technique is to adjust the position linearly along the velocity vector, and the velocity linearly according to forces like gravity or drag. The result is a stepwise-linear approximation, the precision of which depends on the sampling frequency. Another common technique is to do collision detection only once per time step, and again the result depends on the frequency of steps. It is important, then, that the step function is called at a reasonably high frequency.

We could leave students to resolve this themselves, by dividing time steps into multiple finer steps, if necessary, in their step implementation. However, imposing that burden would violate our first requirement: not requiring any discipline from the user. Therefore, we have to ensure that the step function is called often enough, even if there is no user event for a while.

In simulations and interactions, the implemented behavior is to evaluate the step function as quickly as possible between animation frames. Thus, simulations running on faster computers may take smaller steps and be more accurate. The need for eventual consistency precludes this strategy here. Instead, the desired step length for collaborationOf is defined globally and set to one-sixteenth of a second:

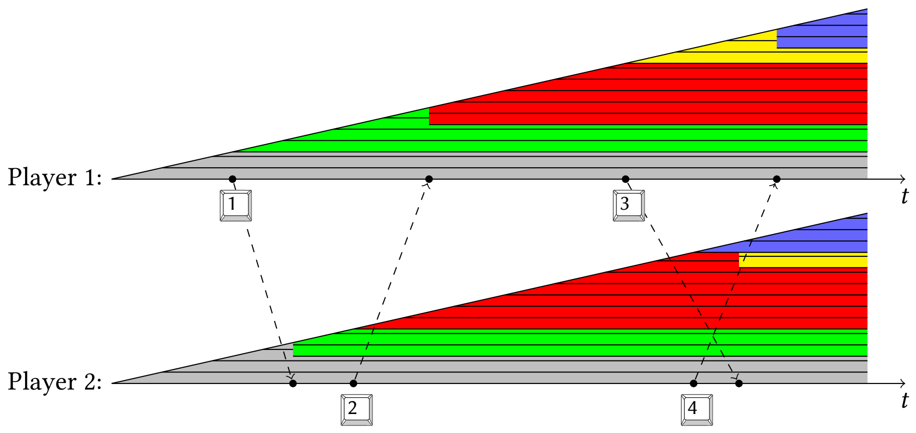

Fig. 8. The evolution of a simple multi-player program with latency in the messages

```
gameRate :: Double
gameRate = 1 / 16
```

We can obtain the desired resolution by wrapping the student's step function in one that iterates step on time steps larger than the desired rate:

```
gameStep :: Game world ⇒ Double → world → world
gameStep dt world | dt ⩽ 0          = world
                  | dt > gameRate = gameStep (dt − gameRate) (step gameRate world)
                  | otherwise      = step dt world
```

Replacing step with gameStep in the implementation of currentState and applyEvents above yields a correct solution.

To see this code in action, we construct the following program: As the time passes, a column grows on the screen, from bottom to top. Initially, it is gray. When a player presses a number key, the column begins to grow in a different color. Additionally, whenever step is called, this current height of the column is marked with a black line.

Because the program output is one-dimensional, we can use the horizontal dimension to show in Figure 8 how the players' displays evolves over time. The dashed arrows indicate the transfer of each packet to the other player, which is not instant. When a message from the other player arrives, the state is updated to reflect this change. Because this game essentially records its history, these delayed updates result in a "flicker" as the client updates the state. In many cases the effect will be less noticeable than it is here. We can see that the algorithm achieved eventual consistency, as the right edge of the drawing looks identical for both clients.

4.4 Limiting Time Travel

In the course of a game, quite a large number of events occur. As time goes by, the cost of calculating the current state from scratch grows without bound, and will eventually become too large to be completed between each frame, and animations will stop being smooth. Clearly, some of that computation is quite pointless to repeat.

Our message transport guarantees that messages from each client are delivered in order, so that when we receive a message, we know that we have seen all messages from the sender up to that timestamp. If we call this the client's commit time, then we know that no new events will be received before the earliest commit time of any client, which we call the commit horizon. We can now precompute the game state up to the commit horizon, forget all older state and events, and use this as the basis for future state recalculations.

In the following we will explain the data structure and associated operations that CodeWorld uses to keep track of the committed state, the pending events and each player's commit time. The main data type is

```
data Log world = Log {committed :: (Timestamp, world),
                      events    :: [Message],
                      latest    :: [(Player, Timestamp)]}
```

Initially, there are no events, and everything is at timestamp zero:

```
initLog :: Game world ⇒ [Player] → Log world
initLog ps = Log (0, start) [ ] [(p, 0) | p ← ps]
```

When an event comes in, the message is added to events via the public addEvent function.

```
addEvent :: Game world ⇒ Message → Log world → Log world
addEvent (t, p, e) log = recordActivity t p (log {events = events'})
  where events' = sortMessages (events log ++ [(t, p, e)])
```

Then, the client's commit time in latest is updated.

```
recordActivity :: Game world ⇒ Timestamp → Player → Log world → Log world
recordActivity t p log | t < t_old  = error "Messages out of order"
                       | otherwise = advanceCommitted (log {latest = latest'})
  where latest' = (p, t) : delete (p, t_old) (latest log)
        Just t_old = lookup p (latest log)
```

This might have moved the commit horizon, and if some of the messages from the list events are from before the commit horizon, we can integrate them into the committed state.

```
advanceCommitted :: Game world ⇒ Log world → Log world
advanceCommitted log = log {events    = to_keep,
                            committed = applyEvents to_commit (committed log)}
  where (to_commit, to_keep) = span (λ(t, _, _). t < commitHorizon log) (events log)

commitHorizon :: Log world → Timestamp
commitHorizon log = minimum [t | (p, t) ← latest log]
```

The final public function is used to query the current state of the game. Starting from the committed state, it applies the pending events.

```
currentState :: Game world ⇒ Timestamp → Log world → world
currentState now log | now < commitHorizon log = error "Cannot look into the past"
currentState now log = gameStep (now − t) world
  where past_events = takeWhile (λ(t, _, _). t ⩽ now) (events log)
        (t, world) = applyEvents past_events (committed log)
```

This algorithm relies on these assumptions:

(1) The list of players provided to initLog is correct.
(2) For each player, events are added in order, with monotonically increasing timestamps.
(3) The state is never queried at a time that lies before commitHorizon.

The first assumption is ensured by the CodeWorld framework. The second is ensured by using a monotonic time source to create the timestamps, and by using an order-preserving communication channel. The third follows from the fact that every client's own timestamps are always in that player's past, and therefore the argument to currentState is later than the commit horizon.

If one of the players were to stop interacting with the program, that client would not send any messages. In this case, no events can be committed and the list of events to be processed by currentState would again grow without bound. To avoid this, each client sends empty messages ("pings") whenever the user has not produced input for a certain amount of time. When such a ping is received, the addPing function advances the latest field without adding a new event:

```
addPing :: Game world ⇒ (Timestamp, Player) → Log world → Log world
addPing (t, p) log = recordActivity t p log
```

Assuming a bounded input event rate and network delay, this bounds the size of the events field.

More tweaks are possible. In the CodeWorld implementation, we also cache the current state, so that until new events come in, repeated use of it is very cheap. Once an input event from another player comes in, we discard this cached value and recalculate it based on the committed state and the stored events.

5 DISCUSSION

The interface from Section 3 allows the creation of multi-user applications with great ease, and with the algorithms in Section 4, CodeWorld can provide a smooth user experience. But surely there are drawbacks and open problems worth discussing.

5.1 Floating Point Calculation

A dominant concern in the implementation in Section 4 was to guarantee eventual consistency of all clients, so that game states would always converge over time. We achieve that requirement, on the assumption that the code passed to collaborationOf consists of pure functions. This result relies on a strong notion of pure function, though, which requires that outputs are predictable even between instances of the code running on different machines, operating systems, and runtime environments. In this sense, even functions in Haskell may not always be pure!

A notable source of nondeterminism in Haskell is underspecified floating point operations. The Double type in Haskell is implementation-defined, and "should cover IEEE double-precision" [Marlow 2010]. Our interest is limited to the Haskell-to-JavaScript compiler GHCJS [Stegeman and Mackenzie 2017], which inherits the floating point operation semantics from JavaScript. The ECMA standard [ECMA International 2015] specifies a JavaScript number to be a "double-precision 64-bit binary format IEEE 754-2008 value" – which is luckily already a quite specific specification. We are optimistic that the basic arithmetic operations are deterministic, encouraged by reports from professional game developers [Emerson 2009]. However, transcendental functions (exp, sin, cos, log, etc.) are not completely specified by IEEE-754, and different browser/system combinations are allowed to yield slightly different results here.

We tested this with a double pendulum simulation, which makes heavy use of sin and cos in every simulation step. The double pendulum is a well-known example of a chaotic system, and we expect it to quickly magnify any divergence in state. Indeed, after running the program on two different browsers (Firefox and Chrome, on the same Linux machine) for several minutes, the simulations take different paths.

If, however, we use a custom implementation of sin – based on a quadratic curve approximation – the simulation runs consistently. We tested this variant on multiple JavaScript engines, OSs and CPUs, and did not uncover any more consistency issues. The tests confirm again that, apart from inconsistent implementations of transcendental functions, basic floating point operations are reliably deterministic in practice.

We can deploy a fix to transcendental functions in two ways. In CodeWorld's educational mode, where we have implemented a custom standard library, it is easy to just substitute new implementations of these functions. In the plain Haskell variant, however, we would like to allow the programmer to make use of existing libraries, which may use standard floating point functions.

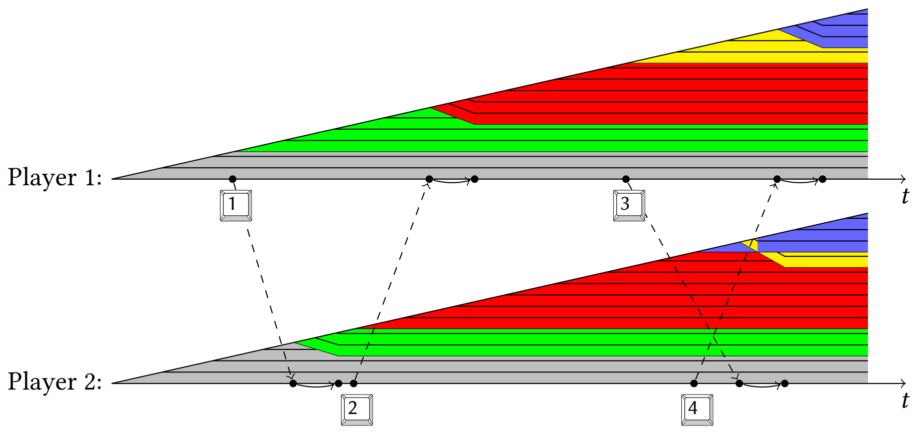

Fig. 9. Smoothing the effect of late events

To achieve this, we can instead replace these operations at the JavaScript level, ensuring that even third-party Haskell libraries are deterministic.

5.2 Interpolating the Effects of Delayed Messages

Another trick in the game programming toolbox is interpolation to smooth out artifacts that result from corrections to the game state. These artifacts can be clearly seen in Figure 8: The moment the message $\boxed{2}$ reaches the first player, the top segment of the growing column abruptly changes from green to red. Similarly, in a first-person shooter, an opponent can appear to teleport to a new location. In this situation, many games would instead interpolate the position smoothly over a fraction of a second. This can introduce new anomalies of its own, but in most cases, it is hoped the result will appear more realistic than the alternative.

By providing an API that is completely abstract in the game state, it seems that we have shut the door on implementing this trick. We lack the ability to look inside the state and adjust positions. In a different setting we could extend the collaborationOf interface with another parameter to interpolate between states, e.g. with type Double → world → world → world.

Surprisingly, though, a form of interpolation is still possible. All that is needed is a sort of change of coordinates. While we cannot interpolate in space, we can interpolate in time! When a delayed event arrives, we initially treat it as if its timestamp is “now" and then slide it backward in time over a short interpolation period until it reaches its actual time.

Usually, the step function is approximately continuous, and as a result, moving an event backwards in time gives a smooth interpolation in the state as well. This can be seen in Figure 9: After the message $\boxed{2}$ arrives at Player 1, the column smoothly changes its color from green to red, from the tip downwards, until the correct state is reached. Like all interpolation, though, anomalies can still happen. This scheme introduces abrupt artifacts as we slide a delayed message past another event with a non-commuting effect. In Figure 9 the second player smoothly integrates the delayed $\boxed{3}$ message, and the top of the column changes color from blue to yellow. But the moment this event is pushed before the local event $\boxed{4}$, the column abruptly changes its color back to blue.

This is an elegant trick to recover the ability to do interpolation. However, it is not clear if interpolation is always the best experience, and a jerky, abrupt update may be preferred for certain games.

5.3 Irreversible Updates

In some cases, the visual artifacts due to delayed messages, whether smooth or jerky, pose a serious problem. Consider, for example, a card game in which both players click to draw cards from the same deck. Suppose player 1 clicks to draw a card first, but the message from player 1 to player 2 arrives after player 2 clicks as well. For a brief moment before the message is received, player 2 sees the top card, even though it ultimately ends up in the first player's hand! This is an example of a case where eventual consistency in the game state is not good enough.

This problem is hard to avoid, given our constraints and the third requirement of responding immediately to local events. It can be mitigated by the game programmer, by adding a short delay before major events such as those that reveal secrets. The delay can sometimes be creatively hidden by animations or effects. This trick dodges the problem as long as network latency is shorter than this delay, but it provides no guarantee. A complete solution to this problem must involve the programmer in a way that is undesirable in our setting, since only the programmer understands which state changes represent a significant enough event to postpone.

6 CONCLUSIONS

By implementing lock-step simulation with client prediction generically in the educational programming environment CodeWorld, we have demonstrated once more that that pure functional programming excels at abstraction and modularity. In addition, this work will directly support the education of our next generation of programmers.

ACKNOWLEDGMENTS

The first author has been supported by the National Science Foundation under Grant No. CCF-1319880 and Grant No. 14-519. We thank Stephanie Weirich, Zach Kost-Smith, Sterling Stein and Justin Hsu for helpful comments on a draft of this paper, as well as Ben Lippmeier and the reviewers for their comments.

REFERENCES

ECMA International. 2015. *ECMAScript Language Specification* (6th ed.). Geneva. http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf

Elijah Emerson. 2009. How to make Box2D more deterministic? (2009). http://www.box2d.org/forum/viewtopic.php?t=1800&start=10#p16662

Jeff Epstein, Andrew P. Black, and Simon L. Peyton Jones. 2011. Towards Haskell in the cloud. In *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*, Koen Claessen (Ed.). ACM, 118–129. DOI: http://dx.doi.org/10.1145/2034675.2034690

Ben Lippmeier. 2017. gloss. http://gloss.ouroborus.net/. (2017).

Simon Marlow (Ed.). 2010. *Haskell 2010 Language Report.*

Forrest Smith. 2011. Synchronous RTS Engines and a Tale of Desyncs. (2011). https://blog.forrestthewoods.com/synchronous-rts-engines-and-a-tale-of-desyncs-9d8c3e48b2be

Luite Stegeman and Hamish Mackenzie. 2017. GHCJS. https://github.com/ghcjs/ghcjs. (2017).

Mark Terrano and Paul Bettner. 2001. 1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond. In *Proceedings of the 15th Games Developers Conference.* http://www.gamasutra.com/view/feature/3094/1500_archers_on_a_288_network_.php