Hochschule für Technik, Wirtschaft und Kultur Leipzig
Leipzig

Fakultät Informatik und Medien
Masterstudiengang Informatik

Masterarbeit
zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

Distributed Systems Extensions for the Dunai FRP Library

Eingereicht von: Julian Götz
Matrikelnummer: 68549

Leipzig 11. Mai 2020

Erstgutachter:	Prof. Dr. rer. nat. Waldmann
Zweitgutachter:	Prof. Dr. rer. nat. habil. Geser

Abstract

Functional Reactive Programming (FRP) offers a declarative way to express reactive systems such as animations, user interfaces and games. Various topics related to FRP like optimization, generalization and debugging were studied. However, the use of FRP in distributed systems has not been investigated extensively.

Focused on the use of the Dunai FRP library implemented in the Haskell programming language, the aim of this thesis is to develop and evaluate a way to apply FRP to distributed systems. A library is implemented to extend Dunai with means to create distributed systems. There is support for the Client/Server network architectural model and algorithms to synchronize applications across a network. As the synchronization of distributed systems has been a topic of research for decades, this thesis explores whether developed ideas, such as Time Warp (Jefferson, 1985), can be expressed in FRP. Additionally, Client Side Prediction (Bernier, 2001) and Dead Reckoning (DIS Steering Committee, 1994) are used to predict server reactions on client-side.

An exemplary application demonstrates the implementation. The application is then evaluated in a performance test and a user test.

Time Warp synchronization has a significant impact on performance. Despite this, the application is playable up to a latency of 100 ms. The result of Dead Reckoning is acceptable, whereas Client Side Prediction is not usable.

The thesis shows that the developed way can be used to run FRP on distributed systems. Further work should focus the performance to enable more complex applications. Moreover, Dead Reckoning can be improved to a smoother result. Non-trivial changes are necessary to make Client Side Prediction usable.

Contents

1 Introduction

Functional Reactive Programming (FRP) has been applied successfully in areas such as game development [CNP03], programming of robots [PHE99; Hud+02] and creation of user interfaces [Cou04]. It is a programming paradigm that regards values that change over time as *Signals*. Signals are defined declaratively and can be combined to create more complex Signals.

However, most applications of FRP involved execution on a single machine. This thesis examines how FRP can be applied to *distributed systems*. A distributed system as defined by Tanenbaum and van Steen, is a set of autonomous, collaborating computers, that appears as one consistent system [TS07, p.2]. As Margara and Salvaneschi [MS18] and Bainomugisha et al. [Bai+13] noted, most existing FRP frameworks do not support the implementation of distributed programs, so this topic remains an open topic for research.

In the context of this thesis, *distributed FRP* means that FRP is executed in parallel on multiple computers that communicate with each other to deliver a single, consistent result. There is an inherent delay of messages when communicating over a network. Due to this, mechanisms need to be used to ensure that exchanged messages are processed in the order they were sent. Such mechanisms are called *synchronization algorithms* [Fuj00, p.51]. The general class of techniques for ensuring a consistent result is called *consistency maintenance mechanisms* [DWM06a, p.225].

An example for distributed FRP is a multiplayer game in which players are simultaneously in a virtual world. For the best possible, consistent experience, all players must be able to equally participate and individual players must not be disadvantaged. Such a disadvantage may arise if commands of a player with low latency are always executed before the commands of other players, although the others issued their commands at the same time or even earlier. Synchronization algorithms can prevent such problems.

The goal of the thesis is the development and evaluation of a library that allows usage of FRP in distributed systems using the programming language Haskell [edi10]. For this purpose we extend *Dunai*, a framework for the implementation of FRP [PBN16, p.33]. Since Dunai is designed in a generalized way, parts of this project have to be implemented in *BearRiver* [PB20a]. BearRiver is an FRP library implemented with Dunai. If we are developing something for BearRiver, this will be explicitly indicated. The library implemented in this thesis features a *Client/Server* architecture and the synchronization algorithm *Time Warp* [Jef85]. Time Warp allows an application to reset to earlier states to process messages at the time they occurred. Therefore, a technique has to be developed that realizes this behavior for Signals in FRP. Furthermore, *Dead Reckoning* [DIS94] is an additional feature to minimize used bandwidth. Dead Reckoning predicts positions of graphical entities by extrapolating known values. Another feature is *Client Side Prediction* [Ber01]. Client Side Prediction can be used to predict the behavior of entities of a distributed application. For implementations related to networking, we use *Cloud Haskell* [EBP11].

The game *Distributed Paddles* serves as an example application. It is inspired by *PONG* [Ata72] and is used to evaluate the performance and quality of the library. Pong simulates table tennis in which two players compete against each other.

The library is evaluated by a performance and a user experiment. There is a loss of performance when using Time Warp, nevertheless Distributed Paddles is playable up to a latency of 100 ms. Dead Reckoning results in acceptable predictions. However, predictions still have to be corrected regularly, causing visible jumps in the presentation. The implementation of Client Side Prediction is not usable in the current state.

The project *FRP for Distributed/Embedded Systems* of the *Tokyo Institute of Technology* counts as related work [Pro19]. However, it focuses the use of FRP in *small-scale*, distributed, embedded systems such as microcontrollers. The researchers define small-scale systems as platforms that are not capable to run full-featured operating systems like Linux [SW16, p.36]. Additionally, they develop the programming language *XFRP* that is compiled into Erlang code. In contrast, this thesis focuses on the implementation in Haskell and also on the execution on more powerful platforms.

Overview of the thesis

The thesis starts with the explanation of FRP in Chapter 2 that also includes an introduction to the Dunai library. Chapter 3 begins with a definition of distributed systems and requirements placed on them. Afterwards the Client/Server architecture is specified and the consistency maintenance mechanisms used in this thesis are described. Cloud Haskell, its structure and functionality for the transmission of messages in a network, is presented in Chapter 4. Chapter 5 introduces the concept of the implemented library and its example application, and then moves on to formulating functional and non-functional requirements for the implementation. After that, technical details of the implementation are presented in Chapter 6. The evaluation of the requirements follows in Chapter 7. Finally, Chapter 8 summarizes the results of the thesis and also gives an outlook on topics that can be based on this work.

2 Functional Reactive Programming

FRP was introduced as *Functional Reactive Animation* (Fran) by Elliott and Hudak in 1997 [EH97] with a focus on the creation of animations. In Fran, continuous, time-varying values of type `a` are *Behaviors* and represented by the polymorphic type `Behavior a`. *Events* constitute values that are defined at discrete points in time, their polymorphic type corresponds to `Event a`.

An example for a Behavior in an animation is a moving circle, as its position is defined at every point in time. Collisions with other moving circles, or keyboard presses are both examples of Events.

Both Behaviors and Events are first-class values which means, for example, that they can be passed as arguments to functions. To generate complex animations, Fran provides a set of combinators to create compositions of Behaviors and Events [EH97, p.264]. Running a program written with Fran means that a composite Behavior is applied to approximations of continuous time [NCP02, p.52].

Fran was first generalized as FRP to other applications, such as robotics, by Wan and Hudak in 2000 [WH00]. In 2001, Courtney and Elliott further developed the paradigm into *Arrowized Functional Reactive Programming* (AFRP) [CE01; NCP02]. It was later released in the library *Yampa* by Hudak, Courtney, Nilsson and Peterson [Hud+02]. Since AFRP is the foundation of Dunai, we will now explain its functionality.

AFRP is centered around the notion of *Signals* and *Signal Functions* (SFs) [NCP02] that are also called *Signal Transformers* [CE01]. Signals correspond to Fran's Behaviors and are values depending on continuous time. The domain of

Time is denoted by the set of positive real numbers $\mathbb{R}$. The polymorphic type `Signal a` equals functions that map values of time to values of type `a`:

Listing 2.1: Definition of Signals [NCP02, p.52].

```
Signal a = Time -> a
```

The value of a Signal at time `t` is called a *sample*.

Signal Functions allow the transformation of Signals and map an input Signal to an output Signal. Their type representation conceptually corresponds to the following:

Listing 2.2: Definition of Signals Functions [NCP02, p.52].

```
SF a b = Signal a -> Signal b
```

So any Signal Function `SF a b` is a function that maps the results `a` of its input to an output of type `b`.

In contrast to Fran, however, Signals are no longer first-class values and just exist implicitly. The decision was made in order to avoid the occurrence of *space-time leaks* [CE01, p. 4]. A space-time leak occurs when all earlier samples are needed to calculate a new one, thus increasing the time and memory consumption without any bound. The incremental calculation of new samples is a solution to prevent space-time leaks [Ell98, p. 289]. By using Signal Functions as first-class values and implementing them using the *Arrow* type class, the possibility of space-time leaks can be eliminated, as proven by Liu and Hudak [LH07].

2.1 The Arrow Type Class

The type class of Arrows was introduced by Hughes in 2000 and forms a generalization of Monads [Hug00, p. 78]. In Haskell the class of Monads has the following definition:

Listing 2.3: Definition of the class of Monads [Hug00, p.69].

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

A polymorphic type `m` forms a Monad if it supports the two operations `return` and `(>>=)`, also known as `bind`. Values of type `m a` are computations with a result type of `a`. The `return` function creates a computation that just returns its argument. The composition of monadic computations `(>>=)`, combines `m a` with a function `a -> m b`, where the result of the former is passed as an argument to the latter, resulting in a new computation `m b` [Hug00, p.69].

Since monadic types `m a` can only be parameterized in their output of type `a`, Hughes introduces the abstract datatype Arrow `a b c`. The type `a b c` means a computation `a` with a parameter of type `b` and a result of type `c` [Hug00, p.77]. That translates to Arrows being parameterized in both their output and input.

This can be extended to Signals and Signal Functions. Signals `Time -> a` can only be transformed in their output, whereas Signal Functions `SF a b` can be transformed in both their input and output [CE01, p.4].

Listing 2.4 shows the definition of the class of Arrows.

Listing 2.4: Definition of the class of Arrows [Hug00, p.79].

```
class Arrow a where
    arr :: (b -> c) -> a b c
    (>>>) :: a b c -> a c d -> a b d
    first :: a b c -> a (b,d) (c,d)
```

`arr` lifts a pure function `b -> c` into the context `a b c` and corresponds to return. At every sample, `arr` returns the result `c` of the function applied to its input `b`.

`(>>>)` is similar to `(>>=)` and denotes the *sequential composition* of Arrow computations. If two Arrows `x` and `y` are connected by `x >>> y`, the result of `x` is used as input for `y` [CE01, p.5f].

The function `first` is defined because `(>>>)` does not allow its input to be retained across multiple calculations. It converts an Arrow from `b` to `c`, to an Arrow with input type `(b,d)` and output type `(c,d)`, where only the first element of type `b` of its input is changed. A second element `d` is returned unmodified [Hug00, p.79]. The functionality of `first` is also called *widening* of an Arrow [CE01, p.6]. If a type implements the Arrow class, the above mentioned functions `arr`, `(>>>)` and `first` are sufficient for the definition.

The supplementary class `ArrowLoop` supports the implementation of *feedback cycles*. A feedback cycle means that part of an output of an Arrow is redirected as an input to the Arrow itself [CE01, p.6]. Listing 2.5 shows the definition of the `ArrowLoop` class.

Listing 2.5: Definition of the ArrowLoop class [CE01, p.6].

```haskell
class Arrow a => ArrowLoop a where
    loop :: a (b, d) (c,d) -> a b c
```

In the `loop` combinator, the output `d` of its argument Arrow is passed back into itself. The resulting Arrow hides the fact that it uses a feedback cycle.

2.2 The Dunai FRP Library

Many incarnations of FRP were published, like *netwire* [Söy18], *reactive-banana* [Apf19] and *reflex* [Tri20]. They offer different models for the treatment of time or functions with side effects and focus different domains of application. Due to the *conceptual similarity* between the various libraries, Perez, Bärenz and Nilsson published Dunai in 2016 [PBN16]. The objective was to find an extensible common denominator, that is capable of supporting the different models for managing time and side effects as well as covering each of the application areas [PBN16, p.33].

2.2.1 Monadic Stream Functions

Perez et al. generalize AFRP's Signal Functions to *Monadic Stream Functions* (MSFs). Monadic Stream Functions are implemented in the polymorphic datatype `MSF`:

Listing 2.6: Definition of Monadic Stream Functions [PB20d].

```haskell
  data MSF m a b = MSF { unMSF :: a -> m (b, MSF m a b) }
```

MSFs are parameterized with a monadic type `m`, an input type `a` and an output type `b`. To execute an MSF, the constructor includes the function `unMSF`. When applied to an input of type `a`, `unMSF` returns a monadic computation in context `m`, that returns a result of type `b` and a continuation.

This continuation is used to process multiple inputs with an MSF. So step by step, starting with the MSF, the returned continuations are applied to inputs to calculate a consecutive set of monadic results. Since the type `MSF` implements the

Arrow class presented in Chapter 2.1, the Arrow operators are available for usage [PBN16, p.36]. The monadic type `m` allows performing monadic computations in each calculation of a sample.

To lift a monadic computation of type `a -> m b` into the context of MSFs, the function `arrM` [Per18, p.96:5], formerly `liftS` [PBN16, p.35], is used:

```
arrM :: Monad m => (a -> m b)-> MSF m a b
```

`arrM` converts a function, of type `a` to a result of type `b` in the monadic context `m`, into an MSF in the context `m` with input type `a` and result type `b`. Its argument is applied on each sample.

The following section describes how different Monads are combined in Dunai. Afterwards a practical example of the Reader Monad is given. The Reader Monad allows functions to access values of an environment.

2.2.2 Combining Monads

Monad Transformers combine the effects of different Monads. Their class is defined as follows:

Listing 2.7: Class of Monad Transformers [Jon95, p.132].

```
class MonadT t where
  lift :: Monad m => m a -> t m a
```

A Monad Transformer of type `t` is a Monad itself, and forms an extended Monad `t m`, where it is able to embed a computation of another monadic type `m` [Jon95, p.132]. The function `lift` takes a monadic computation `m a` and embeds it into the Monad Transformer `t`, thus resulting in a computation of type `t m a`. For more information on combination of Monads, refer to [Jon95, p.131ff], [KW92] and [LHJ95]. To embed MSFs of type `MSF m a b` into MSFs `MSF (t m)a b`, Dunai provides the function `liftTransS` [PB20c]:

```
liftTransS ::
(MonadTrans t, Monad m, Monad (t m))=> MSF m a b -> MSF (t m)a b
```

We will give an example usage of this function in the next section.

2.2.3 Monads in Monadic Stream Functions

In an MSF, different effects can be achieved depending on the Monad used. Suppose the design of an MSF that moves a point in one-dimensional space, starting from a position x0. Without a monadic effect it could be implemented as follows:

Listing 2.8: Moving a one dimensional position to the right, adapted from [PBN16, p.36].

```haskell
positionToRight :: Monad m => Int -> MSF m () Int
positionToRight x0 = count >>> arr (\dt -> x0 + dt)
```

The MSF `positionToRight` receives `x0` as an argument and increases it by 1 on each invocation. The implementation uses `count` that is provided by Dunai [PB20e]. It increases a counter by 1 on each sample, returning 1 on the first sample, 2 on the second and so on. The current value is passed to an Arrow, that sums it up with the initial value of `x0`. Everytime `positionToRight` is called, an argument for `x0` needs to be passed to the MSF.

The Reader Monad enables functions to access values of an environment. Therefore, in this example, `x0` does not need to be explicitly used as an argument:

Listing 2.9: Moving a one dimensional position to the right using the Reader Monad, adapted from [PBN16, p.36].

```haskell
positionToRight :: Monad m => MSF (ReaderT Int m) () Int
positionToRight = liftTransS count
  >>> arrM (\dt -> fmap (\x -> x + dt) ask)
```

`positionToRight` is now an MSF in the context of `ReaderT Int m`. `ReaderT` is the Monad Transformer equivalent to the Reader Monad. The context `ReaderT Int m` shows that an environment with a value of type `Int` is available, where `m` allows embedding other Monads. Since a monadic computation is now required in the implementation, `arrM` is used. The function `(\dt -> fmap (\x -> x + dt) ask)` is passed to it. `ask` is a monadic computation that returns the current value of the environment. Its signature is as follows:

```haskell
ask :: Monad m => ReaderT r m r
```

A computation of type `ReaderT r m a` means that an environment of type `r` is available and the result type of the calculation is type `a`. The type `m` can be another embedded Monad. In this case, the type of ask therefore is
`ask :: Monad m => ReaderT Int m Int`. The function `fmap` applies a function to

the result value of a monadic computation. Here `fmap` calculates the sum of the value of the environment, denoted by `x`, with the current sample of `count`. Finally, the Arrow created with `arrM` has the following type:

```
arrM (\dt -> fmap (\x -> x + dt)ask):: Monad m => MSF (ReaderT Int
m)Int Int
```

While the type of `count` corresponds to the following:

```
count :: Monad m => MSF m ()Int
```

To lift `m` into the context `ReaderT Int m`, `liftTransS` is used, see Section 2.2.2.

For additional applications of monadic effects in MSFs, refer to [PBN16]. For instance, the `Writer` Monad can be used to log messages, or a modifiable state can be introduced with the State monad.

2.3 The BearRiver FRP Library

Since Dunai intents to be a general framework for the implementation of FRP, Perez, Bärenz and Nilsson developed the library *BearRiver*. BearRiver serves as a replacement of the library Yampa [PBN16, p.42]. Its type of Signal Functions is implemented through MSFs. Signal Functions receive on each sample the elapsed time since the last sample, called *delta time*, as values of the type `DTime`, using the transformer variant of the Reader Monad:

Listing 2.10: Type definition of Signal Functions in BearRiver [PB20b].

```
type SF m = MSF ClockInfo
type ClockInfo m = ReaderT DTime m
type DTime = Double
```

Events

Events are used in AFRP to model values that occur at discrete times. They are not associated with any duration and are only defined at certain points in time [NCP02, p.54]. Their definition is as follows:

Listing 2.11: Definition of Events [PB20b].

```
  data Event a = Event a
```

```
                   | NoEvent
```

Events resemble Haskell's type of *Maybe* values [edi10, p.205], as they either contain a value of type `a` or not. Signals carrying event values are called *Event Streams*, while Signal Functions resulting in Events are called *Event Sources* [NCP02, p.54].

The `edge` function is an example of an Event Source, by creating events when its input transitions from false to true.

`edge :: Monad m => SF m Bool (Event ())` [PB20b]

2.4 Executing FRP

Typically, programs written in FRP require side-effectful functions to capture input, such as keyboard presses, and to process output, such as the visual rendering of states. Processing input to output is done by executing a composite SF or MSF. These steps are realized by functions called `reactimate`. Both Dunai and BearRiver provide different variants of the function.

Dunai's `reactimate` is a generalized function that focuses on monadic effects and does not impose any structure [PBN16, p.42]:

Listing 2.12: Signature of `reactimate` in Dunai [PBN16, p.42].

```haskell
reactimate :: Monad m => MSF m () () -> m ()
```

The argument of type `MSF m ()()` is infinitely evaluated. Any functions with side effects can be executed there, since the type parameter `m` is an arbitrary Monad. The signature of the variant in BearRiver is as follows:

Listing 2.13: Signature of `reactimate` in BearRiver [PB20b].

```haskell
reactimate :: Monad m
=> m a                                -- initial sense
-> (Bool -> m (DTime, Maybe a))       -- sense
-> (Bool -> b -> m Bool)              -- actuate
-> SF Identity a b                    -- sf
-> m ()
```

The fourth argument of type `SF Identity a b` is a composite Signal Function that produces samples. It receives input values of type `a` and returns samples of type `b`. Due to purposely restricting it to Signal Functions in the `Identity` Monad, it can not perform any side effects. The first argument produces the initial input of type `a`, passed to the Signal Function at time $t = 0$. The second argument is called *sense*. Sense produces inputs for times $t > 0$ and also computes the time that has passed since the last call to it, the delta time. Inputs are captured in the type `Maybe a`. If its value equals `Nothing`, the last value that equals `Just a` is used as input. The third argument to `reactimate` is called *actuate*. It receives the last output of type `b` and can perform any side effects with it. The returned value of type `Bool` decides whether to terminate the execution [CNP03, p.10].

The execution of distributed FRP requires the implementation of new variants of `reactimate` to account for communication via a network. We will examine these in Chapter 6.2.2.

3 Synchronization of Distributed Systems

This chapter begins with the definition of distributed systems and their requirement of consistency. Then we will introduce the Client/Server architecture. Finally, we will explore techniques that are used to maintain the consistency of a distributed system.

3.1 Distributed Systems

Tanenbaum and van Steen define distributed systems as follows:

> A distributed system is a collection of independent computers that appears to its users as a single coherent system. [TS07, p.2]

In other words, a distributed system consists of several computers, also called *nodes*, that communicate with each other to act as a unified system. The actual type and characteristics of the computers, as well as the type of communication, is usually hidden from the users. In addition, emphasis is placed on providing users with uniform ways to communicate with the distributed system [TS07, p.2]. In a further definition by Lamport [Lam78, p.558], a non-negligible delay in the transmission of messages between individual nodes is also given as a basic requirement of distributed systems.

Since distributed systems involve complex processes and interactions of several computers, it is important to define the structure and communication between them [TS07, p.33]. For this purpose, components of a system and their interactions are integrated as software architectures. One architecture used in this thesis is the *Client/Server architecture* that is introduced in Chapter 3.3.

A *Distributed Interactive Application* (DIA) is a type of distributed system. *Interactive* denotes that an application is modeled as an input-output process, receiving and processing nondeterministic events, such as user input, via a human-computer interface and then displaying it adequately. In DIAs, input can come from multiple users simultaneously, requiring an application to respond to input and produce output in real time [DWM06a, p.220f]. The state of a DIA therefore changes in response to processed events, with events being annotated with the time of their occurrence, their timestamp. Since the execution of FRP using `reactimate`, see Chapter 2.4, realizes input-output processes, applications that execute distributed FRP therefore classify as DIAs.

An important requirement to DIAs is *consistency*, which we will explain in the following chapter.

3.2 Consistency of Distributed Systems

Consistency means that ideally, at any time throughout a distributed application, users are presented the same state. However, perfect consistency is impossible to obtain due to the inherent delay between sending and receiving of a message over a network [DWM06a, p.221].

The length of the delay, that occurs when a message is propagated from one node in a network to another, is called *network latency* or just *latency* [SKH02, p.88]. In fact, the network latency is recognized as the most critical contributing factor to the consistency of DIAs [DWM06a, p.218]. Another factor influencing consistency is *jitter*. Jitter is defined as the variance, or fluctuation, of latency over time [SKH02, p.88].

Depending on the type of application, there are different demands on the level of latency to maintain consistency. For example, the maximum acceptable latency in military simulations is reported between 100 ms and 300 ms [DIS94]. In the case of real-time games, maximum values of 140 ms to 300 ms are specified [Bei+04; PW02a; HB03].

One kind of error that violates consistency is a *causality error*. Causality errors result from the temporally disordered processing of events [Fuj00, p.51]. An error arises if an event a occurs before an event b, but a is processed after b. This can happen due to latency.

Techniques to create adequate consistency, as appropriate to the type of application, are called *consistency maintenance mechanisms* [DWM06a, p.225]. Consistency maintenance mechanisms are categorized into two classes. *Optimistic* mechanisms on the one hand, perform calculations during a DIA and assume that no causality errors occur. The verification whether errors occurred is done during the execution. In the event of an error, optimistic mechanisms have techniques to restore consistency [Fuj00, p.97].

Conservative mechanisms on the other hand, strictly avoid the disordered processing of events. The objective of conservative mechanisms is then to recognize when it is safe to process an event [Fuj00, p.54]. However, since this causes waiting times due to blocking during execution, conservative mechanisms tend to be inferior to optimistic mechanisms in the context of fast-paced interactive applications [DWM06a, p.225].

3.3 Client/Server Architecture

In a Client/Server architecture, running processes are divided into two groups. The first group are *servers*. Servers are defined as processes offering specific services, for example access to a database. The second group consists of *clients*. Clients send requests to access certain elements of the service of a server [TS07, p.37].

In DIAs, a server is a centralized component for managing an application's state. Servers usually perform the main logical operations of an application. This includes the execution of collision detections between simulated entities. Whenever a client wants to perform an action, it is transmitted to the server that updates the state accordingly and sends the resulting state to the clients [RC12, p.11].

3.4 Time Warp Synchronization

Time Warp synchronization is an optimistic consistency maintenance mechanism that was first proposed by Jefferson in 1985 [Jef85]. It calculates new states of a DIA on incoming events without causing causality errors.

Local Control Mechanism

The act of processing new messages is summarized as the *local control mechanism*. Its following description is derived from [Fuj00, p.98ff].

Several nodes form the basis, which send messages to each other and process them. Every node has a *virtual clock* that ticks *virtual time*. Virtual time consists of values of real numbers $\mathbb{R}$ and strictly grows with every message sent. Furthermore, virtual time does not require a connection to real time. Sent messages are associated with a timestamp, containing the current virtual time at the time of their dispatch.

A node using Time Warp has a list of processed messages p and a list of messages to be processed q, both sorted by timestamps in ascending order.

Let x be the oldest unprocessed message with time t_x, thus the first element of q. Let y be the newest processed message with time t_y, therefore last element of p. When processing x there are several possibilities:

1. $t_x > t_y$: The application advances into the future. It optimistically assumes that no message z with a time $t_z < t_p$ exists, so it processes the message.

2. $t_x \leq t_y$: The time t_x of the message lies in the already processed past. This means that the application is ahead of t_x. If t_x was processed, a causality error would occur. A message of this type is called a *straggler message*.

Therefore, the current state of the application is invalid, since x was not taken into account when processing previous events. Messages that have already been processed must consequently be processed again. This process is called a *rollback*. In a rollback, the state of the application is first reset to time t_x. From now on, all messages are reprocessed until the present time t_y.

Rollbacks invalidate states that were already sent to other nodes. A node sends an *anti-message* for each invalid state. An anti-message is identical to the message to be invalidated, except that it contains a flag to identify it as an anti-message. This allows receiving nodes to detect and delete invalid messages as pairs of anti-message and message.

Global Control Mechanism

Since an application using Time Warp stores previous messages and possibly whole states of an application to enable rollbacks, techniques are needed to limit memory consumption. To achieve this, Jefferson introduced *global control mechanisms* [Jef85, p.417].

As virtual time strictly grows within the nodes, it is possible to dynamically calculate a lower limit for possible target times of rollbacks, known as *global virtual time*. Hence, rollbacks are never performed to a time less than global virtual time. Global virtual time is also used to safely execute irreversible I/O operations [Jef85, p.417].

In this thesis we use another procedure, described in Chapter 6.2. A node using Time Warp sets a constant maximum age for new messages. If a too old message is received, it is declared invalid and will not be processed. Messages and states that exceed the maximum age thus are deleted regularly.

3.5 Dead Reckoning

Dead Reckoning is mainly used in DIAs to reduce bandwidth consumption through fewer necessary communication between processes [Fuj00, p.204], and further to decrease the influence of latency [PW02b, p.84]. It classifies as an optimistic consistency maintenance mechanism and is used in DIAs that contain entities moving around in virtual spaces [DWM06b; SH17].

Instead of sending state in every step of an application, it is sent less frequently. Between states, nodes estimate locations of entities from last received positions, velocities and other information. The estimation is done by extrapolating last reports of position and movement of entities [Fuj00, p.204].

There are different types of *Dead Reckoning Models* (DRM) that solve extrapolation of positions in different ways based on the information that is available. Fujimoto [Fuj00, p.207] lists three of these models which we will explain below.

Let the state of a DIA be defined by a n dimensional, Euclidean vector space V^n. Additionally, the state consists of a set of entities described by position vectors $p \in V^n$. The state depends on the *simulation time*, an abstraction used by the DIA to model time that passes inside the simulation. Simulation time is a totally ordered set of values representing points in time. Given two times t_1 and t_2, $t_1 < t_2$ indicates that t_1 *happened before* t_2, whereas $t_1 > t_2$ indicates that t_1 *happened after* t_2 [Fuj00, p.27f].

Now let DRMs be described by functions $x(t)$, returning positions of entities at time t. Let a position of an earlier time t_0, with $t_0 < t$, be given by x_0. The difference between the position at time t and x_0 is denoted as s. It is the distance covered during the time t_0 to t. This leads to the following function $x(t)$:

$$x(t) = x_0 + d \tag{3.1}$$

DRMs according to Fujimoto [Fuj00, p.207] now calculate d in different ways.

Zeroth-order DRM

Zeroth-order DRM is used when additional information is missing. The distance d will be constantly 0, implying that position x_0 equals the position returned by $x(t)$:

$$x(t) = x_0 \tag{3.2}$$

First-order DRM

First-order DRM assumes that entities are moving in a straight line at a known constant velocity $v_0 \in V^n$ between t_0 and t. Integration of v_0 in respect to time delivers the distance s:

$$s = \int v_0 \, dt \tag{3.3}$$

[Fey10, Equation 8.8, p.8-12] Through substitution of s in equation 3.1 by 3.3, as well as by integration of t_0 to t, results the following function $x(t)$:

$$x(t) = x_0 + \int_{t_0}^{t} v_0 \, dt \tag{3.4}$$

Second-order DRM

Second-order DRM is used when the velocity of objects changes by applying a known constant acceleration $a_0 \in V^n$. The distance s is then obtained by twice integrating a_0 [Fey10, p.8-15]:

$$s = \int \int a_0 \, dt \, dt \tag{3.5}$$

Substitution of s in equation 3.1 with 3.5 and twice integrating from t_0 to t, yields $x(t)$:

$$x(t) = x_0 + \int_{t_0}^{t} \int_{t_0}^{t} a_0 \, dt \, dt \tag{3.6}$$

Note that since these DRMs use the integration of time, they can only be implemented in BearRiver. Dunai does not provide a notion of time, whereas BearRiver does. However, a generalized DRM without a predefined integration will be implemented for Dunai.

3.6 Client Side Prediction

It is important to give participants of a DIA feedback on their actions as soon as possible. As the results of actions in a Client/Server architecture must be computed by the server, the network latency and processing time within the server lies between action and reaction. So another technique is needed to give users direct feedback. For this purpose clients use *Client Side Prediction* [Ber01], also referred to as *Client Predict Ahead* [SO09, p.367f]. Client Side Prediction is also an optimistic consistency maintenance mechanism [SO09, p.368].

Client Side Prediction executes logic of a server whenever a user initiates an action. This allows to provide early feedback without having to wait for the server's results. Once a state is received, errors of the prediction are corrected [SO09, p.367f].

In addition, Client Side Prediction can be used if predictions are required for values that cannot be calculated with Dead Reckoning, namely values that are not available as vectors.

4 Cloud Haskell

Cloud Haskell is a domain-specific language for distributed computations embedded in Haskell and was designed in 2011 by Epstein, Black and Peyton Jones [EBP11]. It is inspired by the programming Language Erlang [Eri19] and features a *message-passing* type of communication. In a message-passing model, components of a system communicate with each other by sending messages [EBP11, p.118]. This is opposed to models of *shared memory* where communication happens through shared data structures, providing mutual access to data of other components. In the context of distributed systems these structures are located in simulated shared address spaces [KS08, p.15].

Cloud Haskell is based around *processes* running on *nodes*. A node is a runtime system identified by a unique *NodeId* and an address other nodes use to communicate with. A process is a computation in the *ProcessM* Monad and is identified by a unique *ProcessId* [EBP11, p.119]. In Erlang, a process is a virtual machine that evaluates Erlang functions [Arm07, p.141]. Multiple nodes can be run on the same machine, which enables local testing of distributed applications.

Whereas Erlang exclusively uses message-passing, Cloud Haskell still offers the possibility to use shared-data structures like STM [Har+08] within a single node. However, when communicating over a network, Cloud Haskell's messaging is limited to its built-in message-passing primitives.

If a process a sends messages to a process b, Cloud Haskell guarantees that B receives every message once and all messages in the order they were sent [EBP11, p.119].

4.1 Exchanging Messages

To be able to transmit messages over a network, types to be sent must implement the Serializable type class. All types that implement the Binary and Typeable type classes are of type Serializable. The Binary class converts values into binary format and back. Values of the class Typeable support a function that returns their type representation. Consequently, messages in Cloud Haskell consist of binarized data and the corresponding type representation of the data [EBP11, p.120].

There is a type-safe and a non-type-safe variant for exchanging messages. The `send` function sends a message to a process identified by a `ProcessId`:

LISTING 4.1: Signature of `send` [EBP11, p.119].

```
send :: Serializable a => ProcessId -> a -> ProcessM ()
```

Thus, any value of type `Serializable` can be sent. We then receive a message of a certain type using `expect`:

LISTING 4.2: Signature of `expect` [EBP11, p.119].

```
expect :: Serializable a => ProcessM a
```

Messages that do not match the type are ignored.

Typed channels are used to enable type-safe communication by just allowing sending messages of a specific type. We create a typed channel with `newChan`:

LISTING 4.3: Signature of `newChan` [EBP11, p.120].

```
newChan :: Serializable a => ProcessM (SendPort a, ReceivePort a)
```

Typed channels consist of a `SendPort` and a `ReceivePort`. A `SendPort a` sends values of the type `a` to a `ReceivePort a`. `SendPorts` are serializable and can be sent to other processes, to receive messages from them. This enables a process to receive messages from several others. `ReceivePorts` however, cannot be serialized. The developers of Cloud Haskell have chosen this restriction because messages could be lost while a `ReceivePort` is being sent [EBP11, p.123].

We send serializable messages of type `a` via a typed channel with `sendChan`:

LISTING 4.4: Signature of `sendChan` [EBP11, p.120].

```
sendChan :: Serializable a => SendPort a -> a -> ProcessM ()
```

`receiveChan` is used to receive messages:

Listing 4.5: Signature of `receiveChan` [EBP11, p.120].

```
receiveChan :: Serializable a => ReceivePort a -> ProcessM a
```

4.2 Fault Tolerance

Cloud Haskell also provides functions for monitoring processes and `SendPorts` of a typed channel. Whenever a process or `SendPort` terminates, expected or unexpectedly by an exception, a message is sent to all processes currently monitoring it. With the function `link`, the message is delivered as an asynchronous exception [EBP11, p.121]. When using the `monitor` function, the message is sent as a value of type `ProcessMonitorNotification` via Cloud Haskell's message-passing mechanism [CWV18a].

4.3 The Cloud Haskell Platform

The implementation of Cloud Haskell's ideas was realized in the *Cloud Haskell Platform* [WW17], where the `ProcessM` Monad was renamed `Process`. All the following information about the platform is taken from the documentation available at [WW17].

The Cloud Haskell Platform is a set of libraries for the implementation of distributed applications, with the libraries being divided into three categories. First, there are the *core* libraries. Among them is *distributed-process* [CWV18b], that includes the implementation of the message-passing mechanisms presented in this chapter and process management techniques.

Second, there are so-called *platform* libraries that provide different network architectures. One of them is *distributed-process-client-server* [Wat18], a Client/ Server architecture. The architecture has been tailored for distributed execution of FRP in this thesis, a detailed explanation follows in Chapter 6.1.

Finally, there are libraries of the *network layer*. There is the library *network-transport* [CWV19a], that provides an abstract interface for network communication with the module `Network.Transport`. This interface is used by the core and platform libraries and enables the development of applications independently of the concrete type of communication, such as the *TCP* protocol. The network layer also contains implementations of the interface, such as *network-transport-tcp* [CWV19b], that provides message-passing via the TCP protocol.

5 Concept

After the description of the theoretical basics, the concept of the implemented library is now presented. Starting with a general description of the functionality, functional requirements to it are given in Chapter 5.3, followed by non-functional requirements in Chapter 5.4.

5.1 Concept of the Implemented Library

A library is created to enable the development of DIAs based on the distributed execution of FRP. The Haskell programming language [edi10] serves as the basis for the implementation. A Client/Server architecture enables the creation of server and client applications. Cloud Haskell is used for communication between servers and clients.

Servers start sessions that multiple clients can request to join. A request of this kind is called a `JoinRequest`. When a session is started, FRP will be executed by the server, whereby the current state of the DIA during execution will be communicated to clients regularly, at a rate defined by the user, the `UpdateRate`. States generated by a Server are also called *Updates*, inspired by the game *Half-Life* [Ber01]. The server is the single authority regarding the calculation of states, which means that clients always consider states given by the server as correct.

Similarly, clients run FRP to capture their user's input as well as to process states of the application as reported by the server. Client send inputs to the server that processes them, whereas received states are displayed graphically. Inputs are referred to as *Commands*, also inspired by Half-Life [Ber01].

Figure 5.1 shows an overview of the Client/Server architecture. There are multiple clients connected to a server via a network. Arrows denote bilateral communication between clients and servers. Note that clients do not communicate with each other. Executed operations are given for the first client and omitted for the rest for the sake of clarity.

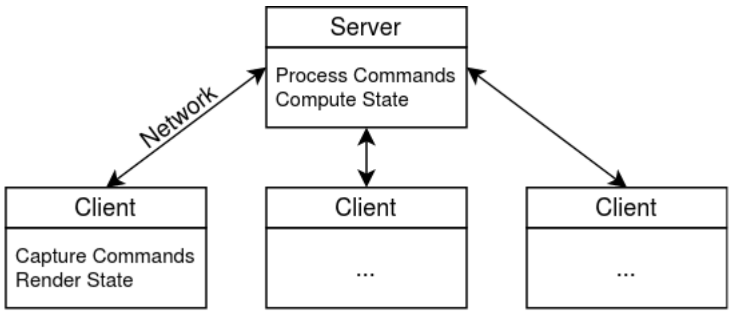

FIGURE 5.1: Overview of the Client/Server architecture.

To execute FRP, the library provides own variants of `reactimate` that hide network interaction from the user. A user of `reactimate` should only choose what data is sent, as opposed to how it is sent. Any input arriving from the network is modeled as an additional input Signal to a user-defined composite Signal Function that produces the states of an application.

If a main program starts a process in Cloud Haskell, its execution always starts in a separate thread [CWV18c]. Consequently, in order to ensure compatibility with graphics libraries, the thread that runs `reactimate` must not be implemented as a `Process` in Cloud Haskell. As an example, the library *GLFW* [Löw20a] throws an error when mouse and keyboard input are accessed outside the main thread [Löw20b].

Furthermore, several consistency maintenance mechanisms are available for use. First, Time Warp synchronization is implemented to avoid causality errors. Since clients always retrieve the latest state from exactly one server, it is only necessary to implement Time Warp specifically for use in servers. As the server always sends the latest available state, it is also not required to implement a technique for sending anti-messages. Newly produced states of the server always invalidate their prior states. As a global control mechanism, the maximum time that can be rolled back can be restricted to a constant limit. Furthermore, the calculation of new states of the application is divided into discrete time steps of uniform length, called *frames*. The length of a time step is configurable by the user in order to reduce amount of required memory when needed. Each frame is also associated with a number, the `FrameNr`, that is incremented with each calculated frame. The `FrameNr` is used as a timestamp for all sent messages to decide on the server whether to execute rollbacks. Clients and servers have to use the same length of time steps to ensure that the application runs correctly. A server informs the client of the current `FrameNr` in each state to guarantee both assume the same `FrameNr`.

Dead Reckoning is implemented to allow states to be sent less frequently. If no current state from a server is available when calculating a client-side state, it is possible to use Dead Reckoning to extrapolate the last state received. To give users of the application direct feedback on their actions, Client Side Prediction is supported by performing server-side simulation immediately without waiting for a response from the server. If a response is then received from the server, the predicted state will be corrected.

The Client/Server architecture and consistency maintenance mechanisms are compatible with the Dunai library. DRMs by Fujimoto [Fuj00, p.207] are an exception, as explained in Chapter 3.5. An additional exception are the implementations of `reactimate`. They prescribe how a program will be executed, whereas the variant in Dunai is too general and does not prescribe a structure, as described in Chapter 2.4. Accordingly, the realized `reactimate` functions are specialized for use in BearRiver.

5.2 Concept of the Sample Application

The application *Distributed Paddles* is implemented as an exemplary application using the features of the library. It serves as an example of usage and is used to evaluate the quality of the library. Distributed Paddles is inspired by the game *PONG* [Ata72] and simulates a game of table tennis between two players. There is a server and client variant of the game. BearRiver is used for implementation, as Dunai is too abstract in this case. SDL2 is used as the graphics library for displaying the game on clients.

The first player to join a session is located as a paddle on the left edge of the field, while the second player is located on the right edge of the field. In the middle of the playing field is a ball that starts flying in the direction of a player at the beginning of the game. The objective as a player is to prevent the ball from getting behind their own paddle. If the ball gets behind the paddle of a player, it returns to the center of the field and starts moving again. Command line arguments are used to decide whether to host a session as a server or join a session as a client. In addition, command line arguments decide which consistency maintenance mechanisms is used.

5.3 Functional Requirements

The functional requirements are formulated separately for general library requirements, general requirements for the Client/Server architecture, specialized server and client requirements, and requirements for the consistency maintenance mechanisms. First, we define general requirements for the implemented library. Note that these do not apply to the exemplary application.

L.1 Compatibility
The implemented library is compatible with Dunai and related libraries, such as BearRiver.

L.2 Independent of graphics libraries
The library should not depend on any graphics library, such as SDL2 or GLFW.

L.3 Compilable by GHC
The library is compilable by the Glasgow Haskell Compiler (GHC) [Gam20].

L.4 Local testing
Running a distributed application locally on a single machine is possible.

L.5 Execution of FRP outside of Cloud Haskell
As described above, it must be possible to run FRP outside the context of Cloud Haskell, that is, outside the `Process` monad.

Now follow general requirements to the Client/Server architecture:

CSA.1 Serializable messages
All types that implement the Serializable type class can be sent over the network. The type class was introduced in Chapter 4.1. This does

not apply to consistency maintenance mechanisms and requirements of `reactimate` functions.

CSA.2 Ordered delivery
All messages between clients and servers are delivered in the order of their sending.

CSA.3 Delivery exactly once
All messages transmitted over a network are delivered exactly once. Lost and duplicated messages are prevented.

CSA.4 Transport layer independency
The Client/Server architecture does not enforce the use of a specific transport layer, such as TCP or UDP.

There are additional specific requirements to server applications:

S.1 Servers execute FRP
There is a `reactimate` function that servers use to execute FRP.

S.2 Servers send states
Servers have means to send the states of an application to clients.

S.3 Servers receive commands
Servers are able to receive commands of clients.

S.4 Servers decide whether clients can join
Servers have the ability to reject or accept `JoinRequests` of clients.

S.5 Fault tolerance of servers
Servers are notified when a connected client terminates, regardless of whether it was planned or due to failure.

S.6 Extensible servers
By default, requests to connect to a server can be made by clients. A server should be possible to extend to handle additional requests.

Furthermore, specific requirements for client applications arise:

C.1 Clients execute FRP
The library ships with a `reactimate` function that clients use to execute FRP.

C.2 Clients receive states
Clients have means to receive states of an executed application.

C.3 Clients send commands
Clients can send commands to a server.

C.4 Clients join a server
Clients can send a request to set up a connection to a server.

C.5 Fault tolerance of clients
Clients are notified when a server terminates, regardless of whether it was planned or due to failure.

Finally, there are functional requirements for the consistency maintenance mechanisms:

CM.1 Time Warp
There is a working implementation of the Time Warp synchronization algorithm.

CM.2 Rollback any MSF
It is possible to rollback any MSF that does not perform side effects.

CM.3 Time Warp prevents causality errors
Using Time Warp synchronization prevents causality errors.

CM.4 Dead Reckoning
All DRMs from Chapter 3.5 are implemented.

CM.5 Client Side Prediction
There is a function to support Client Side Prediction.

5.4 Non-Functional Requirements

Next we specify non-functional requirements. They include general requirements to the quality of consistency maintenance mechanisms, as well as specific requirements to the quality of Dead Reckoning and Client Side Prediction. There are also further requirements to the performance of the project. All requirements mentioned here are evaluated in Chapter 7 using Distributed Paddles in a user experiment and a performance test.

First follow general requirements regarding the playability:

G.1 Playability: Time Warp
The use of Time Warp synchronization leads to an improved playability.

G.2 Playability: Maximum latency
The example application should be playable up to a latency of about 300 ms

using all consistency maintenance mechanisms, according to the maximum values for latency given in Chapter 3.2.

Next are requirements to the quality of the DRMs:

DR.1 Dead Reckoning: Stutter-free movement of the opposing paddle
Dead Reckoning enables an accurate, stutter-free movement of the opposing paddle.

DR.2 Dead Reckoning: Stutter-free movement of the ball
Dead Reckoning enables an accurate, stutter-free movement of the ball.

Then we give requirements for the quality of Client Side Prediction:

CSP.1 Prediction: Instant feedback
Client Side Prediction provides instant feedback on actions.

CSP.2 Prediction: Stutter-free
Client Side Prediction enables an accurate, stutter-free movement of the controlled paddle.

Finally, there are additional requirements to the performance:

P.1 Performance: Server
Performance of servers when using Time Warp suffices.

P.2 Performance: Client
Performance of clients when using consistency maintenance mechanisms suffices.

6 Implementation

The library was implemented as a *Cabal* project [Cab20]. All source code is located in the project directory `distributed-frp-dunai`. Sources of the library are in the directory `src`, tests to satisfy the requirements are in the directory `tests`. The sample application *Distributed Paddles* is in `distributed-paddles`.

In the following, we describe all modules that are relevant for users of the library. They export all functions necessary to perform distributed FRP. In addition, the documentation of the modules generated with Haddock [Mar14] is in the project directory in the folder `doc` in the file `index.html`.

The Client/Server architecture is implemented in the modules `Network.Server` and `Network.Client`. The beginning of Chapter 6.1 explains the realized structure and procedures of the Client/Server architecture, while Chapters 6.1.3 and 6.1.4 deal with technical details of the server and client implementations. Types and functions used by both servers and clients are provided in module `Network.Common`.

Functions for using Time Warp synchronization are in the modules `Data.MonadicStreamFunction.Network.TimeWarp`, and `FRP.BearRiver.Network.Reactimate`. While `Data.MonadicStreamFunction.Network.TimeWarp` provides methods to roll back individual MSFs, `FRP.BearRiver.Network.Reactimate` features a variant of `reactimate` that uses Time Warp synchronization. The description of the implementation follows in Chapter 6.2.

Functions related to Dead Reckoning and Client Side Prediction are in the modules `Data.MonadicStreamFunction.Network.Prediction` and `FRP.BearRiver.Network.Prediction`. Details on the implementation of Dead Reckoning follow in Chapter 6.3, while Client Side Prediction is covered in Chapter 6.4.

6.1 Client/Server Architecture

The Client/Server architecture offers the possibility to implement client and server applications. The objective of the architecture is to implement all network-related aspects separately from FRP, in order to achieve modular, testable components.

Servers provide an interface that receives and processes requests from clients. By default, clients can submit two types of requests, although the set of possible requests is extensible by users of the architecture. On the one hand, a request for participation, called `JoinRequest`, can be issued. On the other hand, commands to be executed can be sent to the server.

Clients however do not offer a separate interface. To allow a server to send application states to a client, a typed channel is established between them to ensure the type-safe transmission of states. Typed channels were previously explained in Chapter 4.1.

The architecture supports two different procedures, establishment of connections between clients and servers and continuous distributed execution of FRP.

6.1.1 Establishing Connections between Clients and Servers

Figure 6.1 shows the process of establishing connections. For the representation, *Business Process Model Notation*(BPMN) [OMG13] is used. On the left side is the flow of a server, while on the right side is the flow of a client. The start of the processes begins with a *StartEvent*, represented by a tagged circle. Tasks performed are shown in the diagram as rounded rectangles connected by black arrows, called *SequenceFlows* in BPMN, which illustrate the flow of tasks [OMG13]. A diamond is shown, if an exclusive case distinction is made during the transition between consecutive tasks. Exactly one outgoing path is taken that depends on the result of the distinction. Communication, or exchanging messages over a network, is represented by dashed arrows. BPMN calls this a *MessageFlow*, where the direction of the arrow indicates sender and recipient [OMG13]. A circle with a bold border represents *EndEvents*, used to indicate the end of a process [OMG13].

The function `startServerProcess` in the module `Network.Server` starts and initializes a server. Servers are implemented as concurrent processes in Cloud

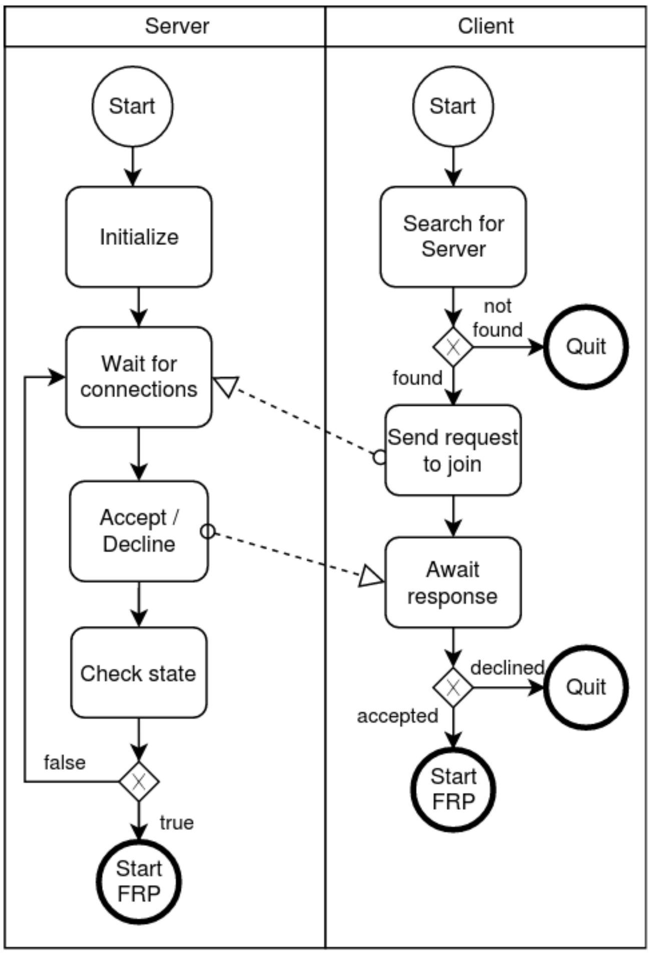

FIGURE 6.1: Procedure of connection establishment between a client and a server.

Haskell and are therefore computations in the `Process` monad. During initialization, a sub-process is started that provides the interface for communication with clients. This is followed by waiting for incoming `JoinRequests`. On an incoming `JoinRequest`, a check is performed whether the request is accepted. A function defined by the library user can be used for this purpose, by default, all requests are accepted. This way, for example, the number of connected clients can be limited.

`JoinRequests` must contain the `SendPort` of the typed channel used to transmit the states of the application. If the request is accepted, the client and its `SendPort` is added to the internal state of the server. This allows the server to use the `SendPort` to send application states. Additionally, monitoring of the `SendPort` is set up, before reporting the result of the request to the client. If the client and the corresponding `SendPort` terminate, the server is notified accordingly.

Because the server runs concurrently, the library provides a function that blocks a calling thread until the server's state meets a certain condition. For example, it is possible to wait with the execution of the main program until enough clients are connected. Figure 6.1 represents this operation by the task *Check State*. When the condition is finally met, the execution of distributed FRP begins.

We will now describe how clients establish a connection. Figure 6.1 shows this procedure on the right-hand side. Afterwards, we will explain the process of distributed execution of FRP in Chapter 6.1.2. A client is started with the function `startClientProcess` in the `Network.Client` module. Just like servers, clients are concurrent processes running in Cloud Haskell. However, before calling the function, the server must be searched for in order to find its `ProcessId`.

If the search is successful, a typed channel is created to transmit states. Then a `JoinRequest` is issued with the corresponding `SendPort` of the channel and the client waits for the response of the server. If the server rejects the request, an error is returned. If it accepts, the execution of distributed FRP starts. Since the server's interface is extensible, it is possible to extend the process by performing other activities instead of transitioning directly to Figure 6.2.

6.1.2 Distributed Execution of FRP

Figure 6.2 shows the flow of distributed execution of FRP using BPMN. The left side of the diagram depicts the flow of a server, while the right side depicts the flow of a client. Both processes are implemented by specialized variants of `reactimate`. The figure focuses on aspects related to the network, so only parts of the loop are visible that are relevant to the communication. We examine both loops in detail in Chapter 6.2.2.

At the beginning of each loop the server reads incoming commands. The polymorphic data type `CommandPacket a` in module `Network.Common` represents commands. A `CommandPacket` contains the `ProcessId` of the sending client, the `FrameNr` of the frame of occurrence and the actual data of type `a`. Furthermore, a `CommandPacket` is serializable if its data `a` is also serializable. The task *FRP* in

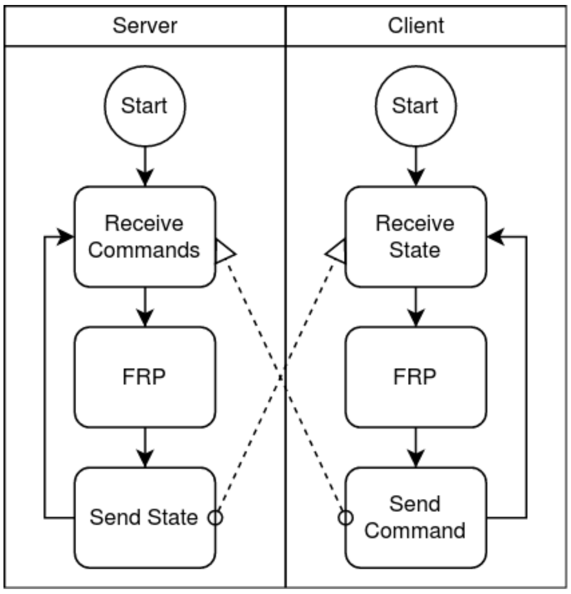

Figure 6.2: Procedure of distributed execution of FRP.

Figure 6.2 processes commands after they were received. This task applies FRP to execute the main logical operations of the application and thus generates a new state. Afterwards the task *Send State* sends the state to all connected clients and their `SendPorts` by using the polymorphic data type `UpdatePacket a` in module `Network.Common`. Analogous to a `CommandPacket`, an `UpdatePacket a` contains the `FrameNr` of the frame of occurrence, the state of type `a` and is serializable if its state is serializable. After having sent an `UpdatePacket`, the loop starts again with the reading of the received commands.

Clients check at the beginning of the loop whether state of the server has arrived, which is then passed to the task *FRP*. The task `FRP` applies FRP to display the latest `UpdatePacket` and to detect user inputs. If desired, Dead Reckoning and Client Side Prediction are also applied there. After executing FRP, the user input is sent to the server as a `CommandPacket`, followed by the loop restarting to receive incoming state.

6.1.3 Implementation of Servers

In this chapter we examine technical details of the server implementation. As already mentioned, servers are implemented as concurrent processes in Cloud

Haskell. To access the server outside the context of Cloud Haskell, we can not use message-passing mechanisms like typed channels, as stated in Chapter 4. For this reason, the function `startServerProcess` returns a record of type `LocalServer a b`. A `LocalServer` serves as an interface to interact with a locally started server. The polymorphic type `a` is the type of commands, while `b` refers to the state of the application. Listing 6.1 shows the definition of the corresponding data type.

Listing 6.1: Definition of the record `LocalServer a b`.

```haskell
data LocalServer a b = LocalServer {
  pidServer :: ProcessId
  , pidApiServer :: TMVar (Either SomeException ProcessId)
  , sendVar :: TMVar ([(SendPort (UpdatePacket b), UpdatePacket b)])
  , readQueue :: TQueue (CommandPacket a)
  , stateServer :: TVar (ServerState b)
  }
```

The field `pidServer` contains the process identifier of the main process of the server and is used to terminate it. A separate `Process` starts the server's interface to receive requests. The `TMVar` `pidApiServer` is initially empty. Once the interface has started, the variable contains either the corresponding `ProcessId` of the interface or an exception if an error occurred during startup. Since `startServerProcess` is implemented as a non-blocking operation and the interface is started concurrently, the value of `pidApiServer` is therefore not yet available at the time of creation of a `LocalServer`. We use `sendVar` to send `UpdatePackets` to clients. A list of tuples is passed to the variable, where the first element is the `SendPort` of the recipient and the second element is the `UpdatePacket` to send. A subprocess of the server retrieves, sends and then empties the contents of the variable at the specified `UpdateRate`. We use `readQueue` to receive commands. Each time a `CommandPacket` arrives, another subprocess adds it to the `readQueue`. Connected clients are accessed via `stateServer`, where `ServerState` is a type synonym for a list of clients. Since the variable always contains a value, an empty list at server startup, the type `TVar` was chosen.

To avoid library users having to design their own functions for interacting with STM structures of a `LocalServer`, there are the functions `writeState` and `receiveCommands`. The function `writeState` puts, if `sendVar` is empty, a new list of `UpdatePackets` into the variable. If `sendVar` already contains a list, the function replaces it since just the current state of the application is relevant for clients. The function `receiveCommands` returns all elements of the `readQueue` and empties it.

For the implementation of the interface the package *distributed-process-client-server* [Wat18] was used. Records of type `ProcessDefinition` define the interface to be hosted by servers and are customizable by library users. A `ProcessDefinition` holds functions responsible to handle requests to incoming messages.

6.1.4 Implementation of Clients

The module `Network.Client` contains the implementation of client processes. Similar to server processes, the focus is on the implementation of an abstraction for communication over a network. A client is a `Process` running in Cloud Haskell that receives states of an application and sends commands to a server. We create a client with the function `startClientProcess` that returns a record of type `LocalClient a b`. The polymorphic type `a` is the type of state, whereas the polymorphic type `b` is the type of commands. Similar to a `LocalServer`, a `LocalClient` serves as an interface for interaction with a client process in Cloud Haskell:

LISTING 6.2: Definition of the record `LocalClient`.

```
data LocalClient a b = LocalClient {
                         pidClient :: ProcessId
                       , readVar :: TMVar (UpdatePacket a)
                       , sendVar :: TMVar (CommandPacket b)
                       }
```

Internally, `startClientProcess` creates the processes `receiveStateProcess` and `sendCommandProcess`. Analogous to `LocalServer`, the STM variable `readVar` then contains the latest received `UpdatePacket` from `receiveStateProcess`, while `sendVar` transmits a `CommandPacket` to `sendCommandProcess`. The process `sendCommandProcess` waits for `sendVar` to be written to and is automatically woken up on write operations to send a command to a server and to empty the variable afterwards, `pidClient` is the process identifier of a `LocalClient`.

We use the functions `writeCommand` and `receiveState` to interact with `sendVar` and `readVar`.

Via `writeCommand`, we put a command into the `sendVar` of a given client. If it already holds a value, meaning the value has not been sent yet, the calling thread is blocked until the variable is free again. The function `receiveState` returns the last received `UpdatePacket` or `Nothing`, if the variable `readVar` currently contains no value.

6.2 Time Warp Synchronization

6.2.1 Rollbacks of Monadic Stream Functions

Monadic Stream Functions carry an internal state that can change with every application. An example of a stateless MSF, is any MSF created by lifting a function *f* with the Arrow operator `arr`. If, for example, the function `\x -> x + 1` is transformed with `arr`, a number received as input is mapped to its successor on every application. In contrast, the MSF `count` has an internal state in that it returns the number of its applications, starting at 1 on its first application. Its signature is as follows:

Listing 6.3: Signature of `count` [PB20e].

```
count :: (Num n, Monad m) => MSF m a n
```

The MSF `count` ignores its input and prints the number of its applications. Thus, it returns the following numbers when used four times: $1, 2, 3, 4$.

This means, in order to enable the implementation of Time Warp synchronization, particularly rollbacks of a DIA to earlier points in time, we require a function that allows any MSF to be transformed in such a way that it can reset to earlier states.

Suppose `count` has been applied four times. If the fifth application rolls back to its second application, it returns 2. Therefore, we expect the MSF to jump 3 states back into its past and return: $1, 2, 3, 4, 2$. To achieve this behavior the function `rollbackMSF` was designed:

Listing 6.4: Signature of `rollbackMSF`, module `Data.MonadicStreamFunction.Network.TimeWarp`.

```
rollbackMSF :: Monad m => Natural -> MSF m a b -> MSF m (Natural, a)
    b
```

The function transforms any MSF with input of type `a` and output of type `b`, into an MSF with an additional input n of type `Natural`, where `Natural` consists of values of natural numbers $\mathbb{N}$. Input n selects the target state of a rollback. If $n = 0$, the MSF is applied to its argument of type `a` without any rollback. If $n > 0$, n states are rolled back and the resulting MSF is applied to `a`. The number of internally stored states to roll back to, is limited to a certain number using the first parameter of type `Natural`. We achieve the above output with the following inputs:

```
(0, ()), (0, ()), (0, ()), (0, ()), (3, ())
```

Since `count` ignores its input, an empty tuple is passed as the second element. We can jump to the immediate predecessor of a state with $n = 1$ to repeat previous output.

The implementation of `rollbackMSF` takes advantage of the constructor of MSFs listed in 2.6. As MSFs return a continuation on each application to calculate subsequent values, the returned continuations are buffered in a feedback cycle within `rollbackMSF`. Feedback cycles are part of the Arrow type class and allow the output of an Arrow to be returned to itself in a subsequent application, as explained in Chapter 2.1.

Two further things to note are important here. First, after rolling back, we consider the target of the rollback as the new, most current state. This means, all continuations newer than the target are discarded, so it is not possible to perform a successive rollback into the future to access later continuations. This is due to rollbacks enabling new inputs to an MSF, perhaps leading to different states than before.

Second, a rollback only works if the MSF has no side effects, as these are not trivial to reverse. For example, if a file change occurs, a rollback would change the file a second time. Module `FRP.BearRiver.Network.TimeWarp` contains the function `rollbackSF`, which constrains its argument to SFs free from side effects. Internally we use `rollbackMSF`. The signature of `rollbackSF` is as follows:

LISTING 6.5: Signature of `rollbackSF`, module `FRP.BearRiver.Network.TimeWarp`.

```
rollbackSF :: Natural -> SF Identity a b -> SF Identity (Natural, a)
    b
```

The constraint is done by accepting only SFs parametrized with the `Identity` Monad. This can not be done for `rollbackMSF`, as it would not be compatible with the Reader Monad used in Signal Functions. There the Monad provides the current `DTime`.

The following chapter introduces a variant of `reactimate` for executing distributed FRP with support for rollbacks in BearRiver. Similar to its original, see Chapter 2.4, it only allows SFs without any side effects.

6.2.2 Executing FRP using Time Warp Synchronization

Execution on a Server

Module `FRP.BearRiver.Network.TimeWarp` exports the function `reactimateTimeWarp` that executes FRP using Time Warp synchronization. Listing 6.6 shows its signature.

Listing 6.6: Signature of `reactimateTimeWarp`, module `FRP.BearRiver.Network .TimeWarp`.

```
reactimateTimeWarp
  :: (Monad m, HasFrameAssociation nin)
  => m a                                -- initial sense
  -> (Bool -> m (DTime, Maybe a))       -- sense
  -> (Bool -> b -> m Bool)              -- actuate
  -> SF Identity (a, [nin]) b           -- sf
  -> m [nin]                            -- receive
  -> ((FrameNr, b) -> m any)            -- send
  -> Natural                            -- maximum number of frames
       to save
  -> m ()
```

The parameters `initial sense`, `sense`, `actuate` work analogously to `reactimate`. They represent functions that obtain network-independent input of type `a` and process output of type `b` using side effects.

An additional parameter is `receive`, a monadic action to retrieve network input of type `nin`, such as `CommandPackets`. The `readQueue` of a `LocalServer` is accessed here. The constraint `HasFrameAssociation` specifies that the type of network input nin must be associated with a FrameNr to determine which frame the input is designated for and to execute rollbacks when necessary.

To send states of the application the parameter send is used. Since the focus is on its performed side effects, it can return any value of type any, which will be ignored during the execution. The function receives as its parameter a tuple of the current FrameNr and output b. Thus, the implementation of the function decides how a UpdatePacket is generated from a value of type b. The parameters receive and send can be any monadic computation, therefore they can be used independently of the Client/Server architecture described in Chapter 6.1. The last parameter of type Natural controls how many states, or frames, are stored for rollbacks. The fourth parameter, called sf, performs the logical operations and yields states of the application. It is a composite SF that receives inputs from the network of type nin as an additional input Signal. To enable rollbacks, sf is transformed with rollbackSF.

Note that reactimateTimeWarp does not provide a specific way to set the length of the executed time steps. Users of the function can therefore decide independently how to limit the length. In the example application, for instance, the sense function blocks execution until enough time has elapsed to trigger the calculation of a new frame.

Figure 6.3 shows a diagram depicting the flow within reactimateTimeWarp using BPMN. At the start of the loop, inputs are captured with sense, as well as commands with receive and the current FrameNr. At the start of the loop, sense captures inputs, receive checks for commands and the current FrameNr is determined. The FrameNr starts at 0 and is incremented by 1 with each pass of the loop.

Then, from the list of all unprocessed commands, those to be processed in the current frame n are selected. The list of unprocessed commands consists of new received commands and those that were not processed in the previous frame, which are accessible via a feedback cycle. The feedback cycle furthermore contains all processed inputs of type a. Commands whose frame association is less than or equal to n are permitted for processing.

The SF sf is transformed with rollbackMSF and applied in the task *FRP* with the current FrameNr n, the inputs of sense and the selected commands. The

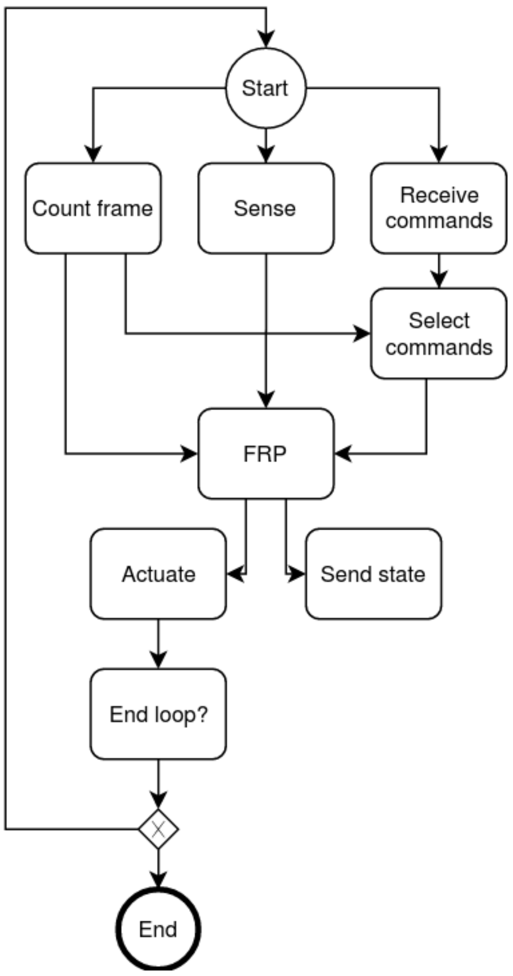

FIGURE 6.3: Reactimate using Time Warp in a server.

unprocessed command with the smallest frame number n_{min} is used to decide whether a rollback is necessary.

If $n_{min} = n$, the commands are up to date, so no rollback is performed. Consequently, the SF `sf` is applied with the input of type `a` and the commands. This case also arises if no commands have been received. The case $n_{min} > n$ can not occur because commands with higher a `FrameNr` have been excluded before.

If $n_{min} < n$, a rollback is initiated by jumping $n - n_{min}$ states into the past. Then prior states from frame n_{min} to $n - 1$ are iteratively recalculated. Stored earlier inputs and commands are used here, while new unprocessed commands are added at their corresponding frames. The last step is to execute $n-1$ to n with the current value of `sense` and commands of the frame, if they already exist.

After executing FRP, `Actuate` is called, which can be used for server-side rendering. Furthermore, `Send State` is invoked and distributes the current state to clients. Whether to exit the loop is determined by the return value of `actuate`, where a value of `True` will terminate it.

Execution on a Client

A separate variant of `reactimate`, called `reactimateClient`, is included in module `FRP.BearRiver.Network.Reactimate` It can be used by clients when a server is using Time Warp. Since servers always provide the current state of the application, messages from the past do not occur. Thus, it is not necessary that clients also use Time Warp. The signature of the function is given in Listing 6.7.

Listing 6.7: Signature of `reactimateClient`, module `FRP.BearRiver.Network.Reactimate`.

```
reactimateClient
  :: (HasFrameAssociation nin, Monad m)
  => m a                             -- initial sense
  -> (Bool -> m (DTime, Maybe a))    -- sense
  -> (Bool -> b -> m Bool)           -- actuate
  -> SF Identity (a, Maybe nin) b    -- sf
  -> m (Maybe nin)                   -- receive state
  -> (FrameNr -> (DTime, a) -> m ()) -- send command
  -> m ()
```

Like `reactimate`, `reactimateClient` receives an initial input of value `a` at time 0, an action to capture successive inputs with `sense` and with `actuate`, an action to process the output of type `b`.

The `sf` parameter allows receiving any network input of type `nin` as additional input. With `receive`, a parameter is provided that fetches network input for use in `sf`. In addition, `send` is an action that sends `CommandPackets` based on the result of `sense` and the current `FrameNr`.

6.3 Dead Reckoning

The objective of Dead Reckoning is to predict values of Signals arriving from the network that are not defined at all times. In general, a `MSF` that implements Dead Reckoning is characterized in that it transforms an input Signal, which does not always carry values, thus of type `Maybe b`, into a continuous Signal of type `b`. This is done by extrapolation of the last defined value. Accordingly, the resulting type is `MSF m (Maybe b)b`.

The library ships with two variants of Dead Reckoning. First, in the module `Data.MonadicStreamFunction.Prediction`, there is the generic function `drm` that allows using Dead Reckoning with the type `MSF`. The second variant are functions specialized for use in BearRiver. They focus SFs and implement the models given in Chapter 3.5. Since Dunai, in contrast to BearRiver, does not contain a notion of time, the models can only be implemented in BearRiver.

To use the generic function `drm`, various conditions must be fulfilled for values of type `b`, that are passed as arguments to the function. Its signature is given in Listing 6.8, where `VectorSpace v a` comprises vectors `v` with values of type `a` [NC18]. First of all, a type `b` must have a position vector of type `v`, marked in the listing as `position`. Depending on the order of the DRM, it is also necessary that type `b` has an additional velocity or acceleration, see parameter `velocity/ acceleration`. The parameter `integration` is an MSF that integrates a vector. Additionally, an initial value of the type `b` must be specified with `initial output` , which is used for extrapolation before the input Signal carries its first defined value. As last requirement, parameter `construct`, it must be possible based on a value `b` and an extrapolated position vector `v`, to create an updated value of type `b` that is used as the output of the MSF.

Listing 6.8: Signature of `drm`, module `Data.MonadicStreamFunction.Prediction`.

```
drm
  :: (VectorSpace v a, Monad m)
  => MSF m v v       -- integration
  -> b               -- initial output
  -> (b -> v)        -- velocity/ acceleration
  -> (b -> v)        -- position
  -> (b -> v -> b)  -- construct
  -> MSF m (Maybe b) b
```

The functions specialized on BearRiver provide direct use of zeroth-order DRM, first-order DRM and second-order DRM. Since BearRiver includes the SF `integral` that integrates a vector `v` over time, there is no longer any need to manually pass an integration function. Additionally, access to position, velocity and acceleration is encapsulated via the type classes `HasPosition`, `HasVelocity` and `HasAcceleration` that provide the functions `getPosition`, `getVelocity` and `getAcceleration`.

The exemplary implementation of second-order DRM by function `drmSecond` is given in Listing 6.9. To call it, we need two arguments, an initial output value and a function to construct extrapolated values. Internally `drmSecond` uses `drm` . We use `integral >>> integral` for integration of values, that is the double integration of an acceleration vector supplied by `getAcceleration`.

Listing 6.9: Implementation of `drmSecond`, module `FRP.BearRiver.Prediction`.

```
drmSecond
  :: (Monad m, VectorSpace v s, HasPosition a v, HasAcceleration a v
     )
  => a                  -- initial output
  -> (a -> v -> a)      -- construct
  -> SF m (Maybe a) a
drmSecond a0 new =
  drm (integral >>> integral) a0 getAcceleration getPosition new
```

6.4 Client Side Prediction

Client Side Prediction refers to executing FRP code of a server locally on clients to predict reactions to user input. A server receives user input of type `a`, processes it into results of type `b` and reports them to the client. Therefore, the server executes an MSF `sf` of the type `MSF m a b`. Clients then receive a discrete Signal of the type `Maybe b` that, for example, is transformed to a continuous Signal using Dead Reckoning, resulting in the type `MSF m (Maybe b)b`. If a client has access to the function `MSF m a b` of the server, it is combined with the discrete signal `Maybe b` to generate a continuous Signal analogous to Dead Reckoning. The result is an MSF of the type `MSF m (a, Maybe b)b` that calculates values of type `b` based on input `a` and corrects its state each time its network input of type `Maybe b` is defined.

This behavior is implemented in the function `predict`, whose signature is given in Listing 6.10. The function takes an MSF as its first parameter that the server uses to process input of type `a` from clients to results of `b`. The function passed in the second parameter is applied as soon as a defined value is received. It applies a new MSF to calculate further values and operates on the base of the defined value.

Listing 6.10: Signature of `predict`, module `Data.MonadicStreamFunction.Prediction`.

```
predict :: Monad m
  => MSF m a b
  -> (b -> MSF m a b)
  -> MSF m (a, Maybe b) b
```

7 Evaluation

This chapter first evaluates the functional requirements of Chapter 5.3. Then we move on to evaluate the non-functional requirements of Chapter 5.4. A performance test and a user test were conducted for this purpose. First, we will describe the concepts of both tests followed by the evaluation of the results.

7.1 Evaluation of Functional Requirements

Now follows a list of all functional requirements and the corresponding proofs of their fulfillment.

Several descriptions refer to unit tests. Comments are given in referenced tests to provide more detailed information about the test procedure. For example, the evaluation of requirement **S.2** uses the test `testSendState` in module `ServerTest`. The source code of the test is shown in the appendix in Listing A.1. The test verifies that writing to the `sendVar` of a `LocalServer` transmits messages. First a `LocalServer` is started. A typed channel is then created to receive messages. Afterwards, several test messages are written to `sendVar` and the `ReceivePort` of the typed channel is checked if it contains them. It also verifies that `sendVar` is empty after sending.

L.1 Compatibility
As Distributed Paddles uses BearRiver and there are unit tests that only use Dunai, the requirement is met.

L.2 Independent of graphics libraries
The library does not have any dependency to a graphics library,
see `distributed-frp-dunai.cabal`.

L.3 Compilable by GHC
The project was compiled by the author using GHC, versions 8.2.2 and 8.8.3.

L.4 Local testing
It is possible by running the executable of Distributed Paddles multiple times. Also, a startup script is given with `run_paddles` that starts the game with one server and two clients.

L.5 Execution of FRP outside of Cloud Haskell
All implemented versions of `reactimate`, as seen in Chapter 6.2.2, are independent of the `Process` monad. The two types `LocalServer` and `LocalClient` are available for communicating with Cloud Haskell.

CSA.1 Serializable messages
The record types `LocalClient a b` and `LocalServer a b` accept any type as command and state types. The functions `Network.Client.startClientProcess` and `Network.Server.startServerProcess` constrain exchanged types to be instances of the Binary and Typeable type classes, which results in Serializable.

CSA.2 Ordered delivery
Cloud Haskell guarantees that messages send are delivered in the order of their sending, which is noted in Chapter 4.

CSA.3 Delivery exactly once
As also mentioned in Chapter 4, Cloud Haskell guarantees that each message reaches its destination exactly once.

CSA.4 Transport layer independency
The library depends only on the package `network-transport`, so it is agnostic to the transport layer, see `distributed-frp-dunai.cabal`.

S.1 Servers execute FRP
There is the function `reactimateTimeWarp` to execute FRP on servers. Its behavior is verified in several tests in module `BearRiverTimeWarpTest`.

S.2 Servers send states
Servers can communicate with clients via typed channels. To send a state to all

clients, the `sendVar` of a `LocalServer` is used. This is also verified in a unit test, see `ServerTest.testSendState`.

S.3 Servers receive commands
Servers receive commands via their interface. Received commands are written to the `readVar` of a `LocalServer`. This is verified in the test `ServerTest.testClientUpdates`.

S.4 Servers decide whether clients can join
A custom routine to decide whether a request to join is accepted can be set in `Network.Server.startServerProcess`. The behavior is tested in the unit test `ServerTest.testJoinRequests`.

S.5 Fault tolerance of servers
As servers monitor clients via Cloud Haskell, this requirement is met. A server maintains the list of connected clients which is updated on changes. In Chapter 6.1.3, we introduced the function `stateServer` to provide access to this list. The behavior is also tested in `ServerTest.testJoinRequests` where connected clients quit after joining a server.

S.6 Extensible servers
An example of an extended server is given in Distributed Paddles, which allows clients to request the initial state of the game. The module `ServerMain` contains the function `reqGameSettings` to retrieve the state via message-passing.

C.1 Clients execute FRP
Clients use FRP by calling the function `reactimateClient` introduced in Chapter 6.2.2. Distributed Paddles uses this function to run clients.

C.2 Clients receive states
When clients submit requests to join a server, they send the `SendPort` of a typed channel to receive states. The `readVar` of a `LocalClient` accesses the latest state. The functionality is confirmed in `ClientTest.testReceiveUpdates`.

C.3 Clients send commands
Clients use the interface provided by servers to send commands, as validated in

`ClientTest.testSendCommands`. To send a command with a `LocalClient`, its `sendVar` is used.

C.4 Clients join a server

Clients use a server's interface to issue `JoinRequests`, which was tested in `ClientTest.testJoinRequest`.

C.5 Fault tolerance of clients

Clients are linked to a server with Cloud Haskell. When a server terminates, so will connected clients. The behavior was checked in the test `ClientTest.testTermination`.

CM.1 Time Warp

Module `BearRiverTimeWarpTest` contains tests related to `reactimateTimeWarp` . It tests whether rollbacks are correctly applied on straggler messages. Also, it tests whether inputs, such as commands, are properly stored in a list so that they can be reused in rollbacks. The list should be sorted according to the frames of occurrence of the input. In addition, it verifies that this list is used during rollbacks to recalculate states. Inputs may only be used for the recalculation of those the frames in which they occurred.

CM.2 Rollback any MSF

By using `rollbackMSF` it is possible to roll back any MSF to earlier states. Its functionality is tested in module `TimeWarpTest`. It is tested whether correct decisions between advancing or rolling back a MSF are made. Also, the correct states should be discarded after a rollback. Note that `rollbackMSF` can not be constrained to MSFs free of side effects, as explained in Chapter 6.2.1. However, `rollbackSF` is a variant which enables this constraint in the context of BearRiver.

CM.3 Time Warp prevents causality errors

As a server using `reactimateTimeWarp` uses rollbacks on straggler messages and then recalculates states, no causality errors can occur.

CM.4 Dead Reckoning

A generalized variant of Dead Reckoning is implemented in module `Data.MonadicStreamFunction.Prediction`, whereas zeroth-order to second-order models are implemented in module `FRP.BearRiver.Prediction`. Every variant is tested in module `BearRiverDRMTest`.

CM.5 Client Side Prediction
There is the function `predict` to support Client Side Prediction. It is implemented in module `Data.MonadicStreamFunction.Prediction` and was validated in module `ClientSidePredictionTest`.

7.2 Concept of the Performance Test

The objective of the performance test is the verification of the Requirements **P.1** and **P.2**. For this purpose, the impact of the consistency maintenance mechanisms Dead Reckoning, Client Side Prediction and Time Warp on the performance during the runtime of Distributed Paddles is determined. To determine the effects, calculated frames per second (FPS) are captured during execution. Since most games run at 30 and 60 FPS, the goal is to achieve an average of 60 FPS [Gre14, p.60].

During the test, two variants of Distributed Paddles are executed 10 times for 60 s each. The test setup consists of two client computers connected in a Local Area Network (LAN) to a computer that runs the server. Random inputs are issued to both clients while testing.

The first variant *NS* does not use any mechanism. Variant two, *TW/DR/CP* evaluates the effect of Time Warp synchronization, Dead Reckoning and Client Side Prediction.

It is expected that performance of the server will decrease with the use of Time Warp as it internally stores past application states and inputs. However, it should still be sufficient. The impact of Dead Reckoning and Client Side Prediction on performance of clients is expected to be low to non-existent.

Additionally, application of profiling will investigate the memory consumption of the server. For this purpose, the server is executed 10 times for $30s$ with and without Time Warp synchronization, afterwards output profiling reports are examined.

7.3 Concept of the User Test

A user test is conducted to evaluate the performance-independent quality of consistency maintenance mechanisms under varying network conditions. The test setup consists of two client computers connected in a LAN to a computer that runs the server. Recruited testers play Distributed Paddles in pairs of two, several rounds against each other. Each round consists of the evaluation of different variants of consistency maintenance mechanisms.

In early test runs, it became apparent that Client Side Prediction does not work as expected, which is why it will not be evaluated in the user experiment. This is due to its accuracy being too low, see Requirement **CSP.1**. A more thorough description of the problem follows in Chapter 7.5.

The first variant *NS* does not use a mechanism. In the second variant, *NS/DR*, Dead Reckoning is evaluated. Variant three, *TW* employs Time Warp synchronization. Variant four, *TW/DR* uses Time Warp synchronization and Dead Reckoning.

After having evaluated one variant, the testers are instructed to complete a questionnaire. After all variants have been evaluated in one round, the latency is gradually increased and all variants are evaluated again. This allows to investigate how the playability changes with increasing latency.

The latency is increased by the program *NETem* [Ber20]. NETem emulates a network interface that allows network conditions such as latency and jitter to be altered synthetically. However, as latency is the main contributing factor to consistency, as stated in Chapter 3.2, only latency is modified in this experiment. However, just latency is modified in this experiment because it is the main contributing factor to consistency, as stated in Chapter 3.2.

Initially, no additional latency is applied. Since the computers are connected via a LAN, the latency is minimal and is only a few milliseconds [SO09, p.339]. After that, the latency gradually increases by 50 ms, 100 ms and 200 ms. Finally, exclusively with variant TW/DR, a test is performed with a latency increased by 300 ms.

According to Pantel and Wolf [PW02a, p.28], a latency of 50 ms and 100 ms is expected to have little impact on the playability when using consistency main-

Table 7.1: The questionnaire used in the User Experiment.

Abbr	Statement	R. Abbr	Requirement
Q.1	The enemy moves without stutters.	**DR.1**	Dead Reckoning: Stutter-free movement of the opposing paddle
Q.2	The ball moves without stutters.	**DR.2**	Dead Reckoning: Stutter-free movement of the ball
Q.5	General impression of the playability	**G.1**, **G.2**	Playability: Time Warp, Playability: Maximum latency

tenance mechanisms. Thus, for variants *TW* and *TW/DR* it is expected that a constant, acceptable playability is achieved, while for variants *NS* and *NS/DR* it decreases. A latency of 200 ms, using consistency maintenance mechanisms, is described as *clearly observable*, but players are able to adapt to it [PW02a, p.28]. As a consequence, it is expected that the game will not be playable without Time Warp synchronization from 200 ms upwards. With Time Warp synchronization it is expected that the playability will decrease but still be acceptable.

Table 7.1 shows the questionnaire. Columns *Statement* and *Abbr* depict the statements and their abbreviations. The columns *Requirement* and *R. Abbr* indicate requirements and their abbreviations, whose fulfillment is assessed by the respective statement.

Each statement from **Q.1** to **Q.4** is rated on a five point scale ranging from 1 to 5, where 1 =“I strongly disagree”, 2 =“I somewhat disagree” , 3 =“Neither agree nor disagree”, 4 =“I somewhat agree”, 5 =“I strongly agree”. Statement **Q.5** is also rated on a five point scale, where 1 =“Very Poor”, 2 =“Poor” , 3 =“Acceptable”, 4 =“Good”, 5 =“Very Good”. The scales were taken from [Bro10].

7.4 Results of the Performance Test

Table 7.2 shows the results of the evaluation of FPS. There are three columns. The first column contains the host for which the respective data in the row was captured. The other two columns present the collected data, divided according

Table 7.2: Results of the performance test, showing achieved average frames per second (FPS), recorded by two clients and one server.

Host	NS		TW/DR/CP	
	Average FPS	σ	Average FPS	σ
Client A	59.7	0.7	59.7	0.8
Client B	60.2	1.2	60.1	0.8
Server	58.9	0.4	46.3	2.0

to the evaluated variant NS and TW/DR/CP. The data is again divided into two columns, the first column reports the average FPS achieved, while the second column shows the standard deviation σ. Additionally, the results of the individual test runs are available in the file `evaluation_results/performance/results.csv`.

In both tested variants, the result of client A with 59.7 FPS is just below the target of 60 FPS. However, it is expected that with additional test runs, the measurement will approach the target. Client B reaches the target with 60.2 FPS and 60.1 FPS.

As expected, Dead Reckoning and Client Side Prediction have no effect on the frame rate. Although client B achieves 0.1 FPS less with Dead Reckoning and Client Side Prediction, it may balance out with more test runs.

In variant NS with 58.9 FPS, the server is just below 60 FPS. However, there is an evident performance degradation when using Time Warp, as the result decreases to 46.3 FPS which is a reduction of 21.4 %. The target is therefore not attained, though a stable frame rate of 30 FPS is possible.

Table 7.3: Results of the server profiling, showing average memory usage and time spent performing garbage collection (GC) when running the example application with and without Time Warp synchronization.

Without Time Warp				With Time Warp			
Memory (MB)	σ	GC (%)	σ	Memory (MB)	σ	GC (%)	σ
4	0	3.2	0.0	201	4	20.5	0.2

Table 7.3 shows the results of the memory profiling. There are two columns, where column *Without Time Warp* indicates values captured without Time Warp,

while column *With Time Warp* indicates values captured using Time Warp. Both columns are again split into two columns and report the maximum used memory in megabytes (MB) and the percentage of the time that was spent by the runtime on garbage collection. The average achieved values and their standard deviation σ are given for both measures.

Without Time Warp, the memory consumption is constantly 4 MB and increases to 197 MB when in use. Furthermore, we see that the time spent on garbage collection increases from 3.2 % to 20.5 %. Since the test system features a memory of 16 GB, it is unlikely that the increased memory consumption is the cause of the drop of performance. According to this, the reason for performance degradation may be that too much time is devoted to garbage collection.

Insufficient performance leads to a loss of playability as the server does not calculate new states fast enough. In the context of the example application, this leads to movements of the paddles and ball being slower than expected. When using Dead Reckoning and Client Side Prediction, clients predict larger movements than calculated by the server. So the predictions are more likely to be incorrect. Since 46.3 FPS are met, the solution is to reduce the frame rate so that clients and servers achieve a common performance.

Finally, the performance of the server is not sufficient to satisfy Requirement **P.1** due to the failure to reach 60 FPS. Requirement **P.1** on the other hand is met with both Clients reaching the defined target.

7.5 Results of the User Test

Since the server lacks performance, the frame rate was limited to 30 FPS in the user experiment to ensure a consistent quality of the application. Table 7.4 shows the results of the user experiment. There are two columns. The first shows the statement, grouped by each evaluated variant, while the second shows the respective scores. The latter is divided into several columns and groups the results according to the applied latency. There are results for latencies from 0 ms to 300 ms. Each result of the individual latencies is again divided into two columns that report the calculated averages of the scores as well as their standard deviation. Addition-

Table 7.4: Results of the user experiment. There is data for each evaluated variant. All data is grouped by the latency during execution.

Statement	Results grouped by latency									
	0 ms		50 ms		100 ms		200 ms		300 ms	
	Average	σ	Average	σ	Average	σ	Average	σ	Average	σ
NS										
Q.1	2.8	1.03	2.2	0.79	1.4	0.52	1.1	0.32		
Q.2	2.6	1.07	2.0	0.82	1.1	0.32	1.0	0.00		
Q.3	1.9	0.74	1.9	0.74	1.5	0.71	1.0	0.00		
NS DR										
Q.1	3.9	0.74	3.4	1.35	2.2	0.63	1.3	0.48		
Q.2	3.6	0.97	3.2	0.79	2.7	0.82	1.8	0.63		
Q.3	2.4	0.84	2.0	0.67	1.3	0.48	1.1	0.32		
TW										
Q.1	2.7	1.06	2.5	0.97	1.4	0.52	1.1	0.32		
Q.2	2.3	0.67	2.1	0.74	1.2	0.42	1.0	0.00		
Q.3	3.3	0.67	3.0	0.67	2.1	0.57	1.4	0.52		
TW DR										
Q.1	3.0	1.05	2.6	0.97	1.4	0.52	1.0	0.00	1.0	0.00
Q.2	3.8	0.92	3.7	0.95	2.8	0.79	1.8	0.79	1.5	0.53
Q.3	4.4	0.70	4.1	0.74	2.9	0.74	1.5	0.71	1.1	0.32

ally, the results of questionnaires are available in the file `evaluation_results/user_test/results.csv`. In the following, we analyze the results.

G.1 The requirement states that Time Warp improves playability and is evaluated with statement Q.3. Values achieved with Time Warp are higher for any latency than values of the NS and NS/DR variants. The requirement is therefore fulfilled. TW/DR achieved the best result with a latency of 0 ms and a score of 4.4.

G.2 The example application is expected to have an acceptable playability under a latency of 300 ms when applying Time Warp and Dead Reckoning. The requirement is not met as the score of Q.3 is 1.1 or "Very Poor". It is playable until 100 ms with an almost acceptable result of 2.9. At 200 ms, it drops to an unacceptable value of 1.5, which is only 0.1 more than TW without Dead Reckoning achieved under the same latency. The evaluation of the Requirements **DR.1** and **DR.2** describes why Dead Reckoning loses effectiveness at higher latencies. Testers identified the lack of responsiveness to input at higher latencies as a strong influence on playability. The time until reactions to inputs are visible increases with latency.

DR.1 Dead Reckoning is expected to enable stutter-free movement of the opposing paddle, which is analyzed using statement S.1. The requirement was only met for variant NS/DR. When comparing NS and NS/DR, Dead Reckoning achieves higher values in all cases. However, the value drops to 2.2 at a latency of 100 ms. This is due to the fact that Dead Reckoning predicts too long paths at higher latencies. Movements are predicted even though they have already ended, but no message has been received yet to end them. Consequently, the accuracy of Dead Reckoning decreases with increasing latency. When comparing the values of TW and TW/DR, Q.1 yields only slightly different values. In fact, Dead Reckoning is not applied to player-controlled entities using Time Warp. Commands of clients modify the velocity vector of the paddle to point to the desired direction. Since commands received by the server always date back in time, the paddles of the players only have a nonzero velocity vector during rollbacks. Thus, the command only affects previous states because the server cannot predict whether a player will still move at the current time. States generated after rollbacks therefore only contain the change of position and no active velocity to use for Dead Reckoning.

DR.2 With Dead Reckoning, movements of the ball should be displayed without stuttering. The requirement is evaluated with Q.2. Testers of variants NS/DR and TW/DR from latency 0 ms to 50 ms tend to agree with Q.2. Starting at 100 ms, the same problem occurs as with evaluation of the previous Requirement **DR.1**. However, higher values were obtained for S.2 than for S.1 at higher latencies. This may be due to the ball, in contrast to the paddles, following a uniform movement. Therefore, Dead Reckoning causes fewer errors. Overall, Requirement **DR.2** can be confirmed, since Dead Reckoning improves smoothness, although not perfectly. In Chapter 8 we will describe a technique to improve the performance of Dead Reckoning in this respect.

CSP.1, **CSP.2** All requirements regarding Client Side Prediction were not met. Client Side Prediction, while providing instant feedback on user input, produced a lot of stutters. Client Side Prediction works by executing commands on client-side without waiting for a server's response. While waiting for the server's reaction, states arrive, where the movement did not happen yet, which set the predicted position back again. So incoming states undo the prediction. This leads to stuttering back and forth between positions and was described by early testers as "deeply disturbing". Chapter 8 explains a technique to prevent this behavior.

8 Summary

The objective of the thesis is to develop and evaluate a way to execute distributed FRP. The thesis started with the introduction of FRP and the libraries Dunai and BearRiver. In Chapter 3, distributed systems were defined and latency was identified as the main problem in maintaining their consistency. To minimize the effects of latency, the mechanisms Time Warp, Dead Reckoning and Client Side Prediction were introduced. A Client/Server architecture based on Cloud Haskell was chosen for implementation. Servers are responsible for computing states of an application. Clients may influence the state via commands that are sent to the server for execution. Servers report changes of state to clients accordingly. Since Cloud Haskell is used, the set of possible requests a server can handle is extensible. Furthermore, Cloud Haskell notifies clients and servers when a connected client or server terminates.

To maintain the consistency of an application, several techniques were implemented. Time Warp was implemented to prevent causality errors that occur when a server does not process commands in time-stamp order, that is, in the order of their occurrence. If a message arrives that would result in a causality error, Time Warp rolls back the current state to the timestamp of the message. The function `rollbackMSF` was developed for this purpose and it reverts any MSF without side effects to prior states. To reduce the impact of latency on clients, Dead Reckoning and Client Side Prediction were implemented. Both offer a way to transform discrete Signals dependent on the network into continuous Signals. An example application was created to demonstrate the implementation.

A performance test and a user experiment were carried out for evaluation with the example application. The performance test measured whether the performance of client and server applications is sufficient to compute 60 FPS. Client applications without Time Warp achieved acceptable performance. It turned out that Time Warp synchronization in server applications leads to performance losses and does not reach 60 FPS frames per second. One reason is that the time spent running Garbage Collection increases significantly. Consequently, more complex applications will not achieve sufficient performance either.

The purpose of the user experiment was to find out whether the consistency maintenance mechanisms are suited to reduce the influence of latency. Since Client Side Prediction severely degraded the playability during early test runs, it was decided not to evaluate it explicitly. With Time Warp and Dead Reckoning the application is playable up to a latency of 100 ms, while without Time Warp unacceptable playability is reached earlier. According to testers, the main problem with higher latencies was the lack of responsiveness to their input.

We identified in Chapter 7.5 several problems of the current implementation of Dead Reckoning and Client Side Prediction. The result of Dead Reckoning is never perfectly smooth. While Dead Reckoning improves the presentation, it does not work with Time Warp for player-controlled entities. In addition, Dead Reckoning does not recognize whether movements are already finished or directions are changed. So with increasing latency the accuracy of Dead Reckoning decreases. Furthermore, Client Side Prediction does not work because its calculations are cancelled and rolled back by incoming states in which a command was not processed yet. We will now describe techniques to mitigate the problems of the implementation.

Future Work

Since the project was evaluated in a local network, an additional test should be carried out in a distributed system with greater distances between components. As a consequence, the network conditions of individual components would vary more. This allows a more precise assessment of the quality of the implementation.

Currently, past states of an application need to be stored in memory with Time Warp. To increase the performance of Time Warp and to reduce its memory requirements, past states can be calculated dynamically. A technique introduced by Perez [Per17] can be used for this.

The delta time, `DTime`, indicates how much time has passed since the last application of an SF and is used to calculate new samples of SFs. It is generally positive as SFs advance into the future. When using a negative `DTime`, SFs can step into the past. This requires the use of special constructions for the development of SFs, for example to undo structural changes [Per17, p.108].

Predictions by Dead Reckoning are corrected immediately on incoming states which leads to visible jumps between positions. To avoid these jumps and achieve a smooth image, positions can be corrected gradually. Such a technique was introduced by Lin, Wang, Wang and Schab [Lin+96] in 1996 and is called *smoothing*. Smoothing works by interpolating between current and corrected positions.

Position History-based Dead Reckoning can be used to represent player-controlled entities. The procedure was introduced in 1994 by Singhal and Cheriton [SC94]. Statements about the velocity of entities are then based on past positions. Thus, changes in position are calculated independently of the velocity reported by the server.

Dead Reckoning is not aware of changes of direction until a new state arrives. In the example application it would be possible to run the logic of the ball locally on clients to predict its movement and collisions. However, in contexts where this is not possible, a hybrid approach of Dead Reckoning and Client Side Prediction may be feasible. Then it might be useful to predict motion by Dead Reckoning and to perform local collision detection by Client Side Prediction. Note that as the motion of player controlled entities is not deterministic, this approach would not work for them.

To perform Client Side Prediction without errors, states, or parts of states that reset predicted values need to be discarded. However, it is not possible to decide whether a state contains the reaction to a command or whether the command was rejected by the server. This requires a *confirmation message* to be introduced. Confirmation messages are sent from the server to clients in response to commands. They indicate whether a command is accepted or rejected. On pending confirmations clients use Client Side Prediction and filter incoming states accordingly. Clients correct their local state when a confirmation message is received. Confirmation messages are also used by Bernier in the Half-Life games [Ber01].

Bibliography

[Apf19] Heinrich Apfelmus. *reactive-banana: Library for functional reactive programming (FRP).* Version 1.2.1.0. 2019. URL: https://hackage.haskell.org/package/reactive-banana (visited on 02/27/2020).

[Arm07] Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* Raleigh, NC, and Dallas, TX: The Pragmatic Programmers, LLC, 2007. ISBN: 978-1-9343560-0-5. URL: http://www.pragprog.com/titles/jaerlang/programming-erlang.

[Ata72] Atari Interactive, Inc. *PONG.* 1972.

[Bai+13] Engineer Bainomugisha et al. "A survey on reactive programming". In: *ACM Comput. Surv.* 45.4 (2013), 52:1–52:34. DOI: 10.1145/2501654.2501666. URL: https://doi.org/10.1145/2501654.2501666.

[Bei+04] Tom Beigbeder et al. "The effects of loss and latency on user performance in unreal tournament 2003®". In: *Proceedings of the 3rd Workshop on Network and System Support for Games, NETGAMES 2004, Portland, Oregon, USA, August 30, 2004.* Ed. by Wu-chang Feng. ACM, 2004, pp. 144–151. ISBN: 1-58113-942-X. DOI: 10.1145/1016540.1016556. URL: https://doi.org/10.1145/1016540.1016556.

[Ber01] Yahn W. Bernier. *Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization.* 2001. URL: https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_

In-game_Protocol_Design_and_Optimization#Lag_Compensation (visited on 05/06/2020).

[Ber20] Bernhard Ehlers. *Bernhard's Homepage. NETem - Network Link Emulator for GNS3*. https://www.bernhard-ehlers.de/projects/netem/index.html. 2020. (Visited on 01/01/2020).

[Bro10] Sorrel Brown. *Likert Scale Examples for Surveys*. 2010. URL: https://www.extension.iastate.edu/Documents/ANR/LikertScaleExamplesforSurveys.pdf (visited on 05/06/2020).

[Cab20] Cabal Development Team. *Cabal: A framework for packaging Haskell software*. Version 3.2.0.0. 2020. URL: https://hackage.haskell.org/package/Cabal (visited on 05/06/2020).

[CE01] Antony Courtney and Conal Elliott. "Genuinely Functional User Interfaces". In: *Proceedings of the 2001 Haskell Workshop*. Sept. 2001.

[Che19] Roman Cheplyaka. *tasty-hunit: HUnit support for the Tasty test framework*. Version 0.10.0.2. 2019. URL: https://hackage.haskell.org/package/tasty-hunit (visited on 05/06/2020).

[CNP03] Antony Courtney, Henrik Nilsson, and John Peterson. "The Yampa arcade". In: *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2003, Uppsala, Sweden, August 28, 2003*. ACM, 2003, pp. 7–18. DOI: 10.1145/871895.871897. URL: https://doi.org/10.1145/871895.871897.

[Cou04] Antony Alexander Courtney. "Modeling User Interfaces in a Functional Language". AAI3125177. PhD thesis. New Haven, CT, USA, 2004.

[CWV18a] Duncan Coutts, Nicolas Wu, and Edsko de Vries. *Control.Distributed.Process*. Version 0.7.4. 2018. URL: https://hackage.haskell.org/package/distributed-process-0.7.4/docs/Control-Distributed-Process.html (visited on 05/09/2020).

[CWV18b] Duncan Coutts, Nicolas Wu, and Edsko de Vries. *distributed-process: Cloud Haskell: Erlang-style concurrency in Haskell*. Version 0.7.4. 2018. URL: https://hackage.haskell.org/package/distributed-process (visited on 01/10/2020).

[CWV18c] Duncan Coutts, Nicolas Wu, and Edsko de Vries. *distributed-process: Cloud Haskell: Erlang-style concurrency in Haskell*. Version 0.7.4. 2018. URL: https://hackage.haskell.org/package/distributed-process-0.7.4/docs/src/Control.Distributed.Process.Node.html#runProcess (visited on 01/10/2020).

[CWV19a] Duncan Coutts, Nicolas Wu, and Edsko de Vries. *network-transport: Network abstraction layer*. Version 0.5.4. 2019. URL: https://hackage.haskell.org/package/network-transport (visited on 03/21/2020).

[CWV19b] Duncan Coutts, Nicolas Wu, and Edsko de Vries. *network-transport-tcp: TCP instantiation of Network.Transport*. Version 0.7.0. 2019. URL: https://hackage.haskell.org/package/network-transport-tcp (visited on 03/21/2020).

[DIS94] DIS Steering Committee. *The DIS Vision: A Map to the Future of Distributed Simulation*. Tech. rep. University of Central Florida - Institute for Simulation and Training, May 1994.

[DWM06a] Declan Delaney, Tomás Ward, and Séamus McLoone. “On Consistency and Network Latency in Distributed Interactive Applications: A Survey - Part I”. In: *Presence* 15.2 (2006), pp. 218–

234. DOI: 10.1162/pres.2006.15.2.218. URL: https://doi.org/10.1162/pres.2006.15.2.218.

[DWM06b] Declan Delaney, Tomás Ward, and Séamus McLoone. "On Consistency and Network Latency in Distributed Interactive Applications: A Survey - Part II". In: *Presence* 15.4 (2006), pp. 465–482. DOI: 10.1162/pres.15.4.465. URL: https://doi.org/10.1162/pres.15.4.465.

[EBP11] Jeff Epstein, Andrew P. Black, and Simon L. Peyton Jones. "Towards Haskell in the cloud". In: *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*. Ed. by Koen Claessen. ACM, 2011, pp. 118–129. ISBN: 978-1-4503-0860-1. DOI: 10.1145/2034675.2034690. URL: https://doi.org/10.1145/2034675.2034690.

[edi10] Simon Marlow (editor). *Haskell 2010. Language Report*. 2010. URL: https://www.haskell.org/definition/haskell2010.pdf (visited on 01/10/2020).

[EH97] Conal Elliott and Paul Hudak. "Functional Reactive Animation". In: *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*. ICFP '97. Amsterdam, The Netherlands: ACM, 1997, pp. 263–273. ISBN: 0-89791-918-1. DOI: 10.1145/258948.258973. URL: http://doi.acm.org/10.1145/258948.258973.

[Ell98] Conal Elliott. "Functional Implementations of Continuous Modeled Animation". In: *Proceedings of PLILP/ALP*. 1998. URL: http://conal.net/papers/plilpalp98/.

[Eri19] Ericsson AB. *Erlang/ OTP System Documentation*. 2019. URL: https://erlang.org/doc/pdf/otp-system-documentation.pdf (visited on 01/14/2020).

[Fey10] Richard P. Feynman. *The Feynman lectures on physics: new millennium edition*. New York, NY, USA: Basic Books, 2010. ISBN: 978-0-465-02562-6.

[Fuj00] Richard M. Fujimoto. *Parallel and Distributed Simulation Systems*. 1st. New York, NY, USA: John Wiley & Sons, Inc., 2000. ISBN: 0471183830.

[Gam20] Ben Gamari. *Home - The Glasgow Haskell Compiler*. 2020. URL: https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization#Lag_Compensation (visited on 05/06/2020).

[Gre14] Jason Gregory. *Game Engine Architecture, Second Edition*. 2nd. Natick, MA, USA: A. K. Peters, Ltd., 2014. ISBN: 978-1-4665-6006-2.

[Har+08] Tim Harris et al. “Composable memory transactions”. In: *Commun. ACM* 51.8 (2008), pp. 91–100. DOI: 10.1145/1378704.1378725. URL: https://doi.org/10.1145/1378704.1378725.

[HB03] Tristan Henderson and Saleem Bhatti. “Networked games - A QoS-sensitive application for QoS-insensitive users?” In: (Sept. 2003). DOI: 10.1145/944592.944601.

[Hud+02] Paul Hudak et al. “Arrows, Robots, and Functional Reactive Programming”. In: *Advanced Functional Programming, 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002, Revised Lectures*. Ed. by Johan Jeuring and Simon L. Peyton Jones. Vol. 2638. Lecture Notes in Computer Science. Springer, 2002, pp. 159–187. ISBN: 3-540-40132-6. DOI: 10.1007/978-3-540-44833-4_6. URL: https://doi.org/10.1007/978-3-540-44833-4%5C_6.

[Hug00] John Hughes. “Generalising monads to arrows”. In: *Sci. Comput. Program.* 37.1-3 (2000), pp. 67–111. DOI: 10.1016/S0167-6423(99)00023-4. URL: https://doi.org/10.1016/S0167-6423(99)00023-4.

[Jef85] David R. Jefferson. “Virtual Time”. In: *ACM Trans. Program. Lang. Syst.* 7.3 (1985), pp. 404–425. DOI: 10.1145/3916.3988. URL: https://doi.org/10.1145/3916.3988.

[Jon95] Mark P. Jones. “Functional Programming with Overloading and Higher-Order Polymorphism”. In: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*. Ed. by Johan Jeuring and Erik Meijer. Vol. 925. Lecture Notes in Computer Science. Springer, 1995, pp. 97–136. ISBN: 3-540-59451-5. DOI: 10.1007/3-540-59451-5_4. URL: https://doi.org/10.1007/3-540-59451-5%5C_4.

[KS08] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. 1st ed. New York, NY, USA: Cambridge University Press, 2008. ISBN: 0521876346.

[KW92] David J. King and Philip Wadler. “Combining Monads”. In: *Functional Programming, Glasgow 1992, Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, UK, 6-8 July 1992*. Ed. by John Launchbury and Patrick M. Sansom. Workshops in Computing. Springer, 1992, pp. 134–143. ISBN: 3-540-19820-2. DOI: 10.1007/978-1-4471-3215-8_12. URL: https://doi.org/10.1007/978-1-4471-3215-8%5C_12.

[Lam78] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Commun. ACM* 21.7 (1978), pp. 558–565. DOI: 10.1145/359545.359563. URL: https://doi.org/10.1145/359545.359563.

[LH07] Hai Liu and Paul Hudak. “Plugging a Space Leak with an Arrow”. In: *Electr. Notes Theor. Comput. Sci.* 193 (2007), pp. 29–45. DOI: 10.1016/j.entcs.2007.10.006. URL: https://doi.org/10.1016/j.entcs.2007.10.006.

[LHJ95] Sheng Liang, Paul Hudak, and Mark P. Jones. "Monad Transformers and Modular Interpreters". In: *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*. Ed. by Ron K. Cytron and Peter Lee. ACM Press, 1995, pp. 333–343. ISBN: 0-89791-692-1. DOI: 10.1145/199448.199528. URL: https://doi.org/10.1145/199448.199528.

[Lin+96] Kuo-Chi Lin et al. "Smoothing a Dead Reckoning Image in Distributed Interactive Simulation". In: *Journal of Aircraft* 33.2 (1996), pp. 450–452. DOI: 10.2514/3.46962.

[Löw20a] Camilla Löwy. *GLFW*. 2020. URL: https://www.glfw.org/ (visited on 01/10/2020).

[Löw20b] Camilla Löwy. *GLFW: Window reference*. 2020. URL: https://www.glfw.org/docs/latest/group__window.html#ga37bd57223967b4211d60ca1a0bf3c832 (visited on 01/10/2020).

[Mar14] Simon Marlow. *Haddock: A Haskell Documentation Tool*. 2014. URL: https://www.haskell.org/haddock/ (visited on 05/06/2020).

[MS18] Alessandro Margara and Guido Salvaneschi. "On the Semantics of Distributed Reactive Programming: The Cost of Consistency". In: *IEEE Trans. Software Eng.* 44.7 (2018), pp. 689–711. DOI: 10.1109/TSE.2018.2833109. URL: https://doi.org/10.1109/TSE.2018.2833109.

[NC18] Henrik Nilsson and Antony Courtney. *simple-affine-space: A simple library for affine and vector spaces*. Version 0.1. 2018. URL: https://hackage.haskell.org/package/simple-affine-space-0.1 (visited on 01/10/2020).

[NCP02] Henrik Nilsson, Antony Courtney, and John Peterson. “Functional Reactive Programming, Continued”. In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell.* Haskell ’02. 565022. Pittsburgh, Pennsylvania: ACM, 2002, pp. 51–64. ISBN: 1-58113-605-6. DOI: 10.1145/581690.581695. URL: http://doi.acm.org/10.1145/581690.581695.

[OMG13] OMG. *Business Process Model and Notation (BPMN), Version 2.0.2.* Object Management Group, Dec. 2013. URL: http://www.omg.org/spec/BPMN/2.0.2 (visited on 04/01/2020).

[PB20a] Ivan Perez and Manuel Bärenz. *bearriver: A replacement of Yampa based on Monadic Stream Functions.* Version 0.13.1.1. 2020. URL: https://hackage.haskell.org/package/bearriver (visited on 02/27/2020).

[PB20b] Ivan Perez and Manuel Bärenz. *FRP.Yampa. Documentation.* Version 0.13.1.1. 2020. URL: https://hackage.haskell.org/package/bearriver-0.13.1.1/docs/FRP-BearRiver.html (visited on 02/27/2020).

[PB20c] Ivan Perez and Manuel Bärenz. *module Data.MonadicStreamFunction.Core.* Version 0.6.0. 2020. URL: https://hackage.haskell.org/package/dunai-0.6.0/docs/Data-MonadicStreamFunction-Core.html (visited on 02/29/2020).

[PB20d] Ivan Perez and Manuel Bärenz. *module Data.MonadicStreamFunction.InternalCore.* Version 0.6.0. 2020. URL: https://hackage.haskell.org/package/dunai-0.6.0/docs/src/Data.MonadicStreamFunction.InternalCore.html#MSF (visited on 01/10/2020).

[PB20e] Ivan Perez and Manuel Bärenz. *module Data.MonadicStreamFunction.InternalCore.* Version 0.7.0. 2020. URL: https://hackage.haskell.org/package/

dunai-0.7.0/docs/Data-MonadicStreamFunction-Util.html#v:count (visited on 05/10/2020).

[PBN16] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. "Functional Reactive Programming, Refactored". In: *Proceedings of the 9th International Symposium on Haskell*. Haskell 2016. Nara, Japan: ACM, 2016, pp. 33–44. ISBN: 978-1-4503-4434-0. DOI: 10.1145/2976002.2976010. URL: http://doi.acm.org/10.1145/2976002.2976010.

[Per17] Ivan Perez. "Back to the Future: Time Travel in FRP". In: *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*. Haskell 2017. Oxford, UK: ACM, 2017, pp. 105–116. ISBN: 978-1-4503-5182-9. DOI: 10.1145/3122955.3122957. URL: http://doi.acm.org/10.1145/3122955.3122957.

[Per18] Ivan Perez. "Fault Tolerant Functional Reactive Programming (Functional Pearl)". In: *Proc. ACM Program. Lang.* 2.ICFP (July 2018), 96:1–96:30. ISSN: 2475-1421. DOI: 10.1145/3236791. URL: http://doi.acm.org/10.1145/3236791.

[PHE99] John Peterson, Paul Hudak, and Conal Elliott. "Lambda in Motion: Controlling Robots with Haskell". In: *Practical Aspects of Declarative Languages, First International Workshop, PADL '99, San Antonio, Texas, USA, January 18-19, 1999, Proceedings*. Ed. by Gopal Gupta. Vol. 1551. Lecture Notes in Computer Science. Springer, 1999, pp. 91–105. ISBN: 3-540-65527-1. DOI: 10.1007/3-540-49201-1_7. URL: https://doi.org/10.1007/3-540-49201-1%5C_7.

[Pro19] Programming Systems Group. *FRP for Distributed/Embedded Systems*. https://www.psg.c.titech.ac.jp/frp_embedded.html. 2019. (Visited on 12/13/2019).

[PW02a] Lothar Pantel and Lars C. Wolf. "On the impact of delay on real-time multiplayer games". In: *Network and Operating System Support for Digital Audio and Video, 12th International Workshop,* *NOSSDAV 2002, Miami Beach, Florida, USA, May 12-14, 2002, Proceedings*. ACM, 2002, pp. 23–29. DOI: 10.1145/507670.507674. URL: https://doi.org/10.1145/507670.507674.

[PW02b] Lothar Pantel and Lars C. Wolf. “On the suitability of dead reckoning schemes for games”. In: *Proceedings of the 1st Workshop on Network and System Support for Games, NETGAMES 2002, Braunschweig, Germany, April 16-17, 2002, 2003*. Ed. by Lars C. Wolf. ACM, 2002, pp. 79–84. ISBN: 1-58113-493-2. DOI: 10.1145/566500.566512. URL: https://doi.org/10.1145/566500.566512.

[RC12] Laura Ricci and Emanuele Carlini. “Distributed Virtual Environments: From client server to cloud and P2P architectures”. In: *2012 International Conference on High Performance Computing & Simulation, HPCS 2012, Madrid, Spain, July 2-6, 2012*. Ed. by Waleed W. Smari and Vesna Zeljkovic. IEEE, 2012, pp. 8–17. ISBN: 978-1-4673-2359-8. DOI: 10.1109/HPCSim.2012.6266885. URL: https://doi.org/10.1109/HPCSim.2012.6266885.

[SC94] Sandeep Singhal and David Cheriton. *Using a Position History-Based Protocol for Distributed Object Visualization*. Tech. rep. Stanford University - Department of Computer Science, Feb. 1994.

[SH17] Jouni Smed and Harri Hakonen. “Compensating Resource Limitations”. In: *Algorithms and Networking for Computer Games*. New York, NY, USA: John Wiley & Sons, Ltd, 2017. Chap. 12, pp. 255–288. ISBN: 9781119259770. DOI: 10.1002/9781119259770.ch12. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119259770.ch12. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119259770.ch12.

[SKH02] Jouni Smed, Timo Kaukoranta, and Harri Hakonen. “Aspects of networking in multiplayer computer games”. In: *The Elec-*

tronic Library 20.2 (2002), pp. 87–97. DOI: 10.1108/02640470210424392. URL: https://doi.org/10.1108/02640470210424392.

[SO09] Anthony Steed and Manuel Fradinho Duarte de Oliveira. *Networked Graphics - Building Networked Games and Virtual Environments*. Academic Press, 2009. ISBN: 978-0-12-374423-4. URL: http://www.elsevierdirect.com/product.jsp?isbn=9780123744234.

[Söy18] Ertugrul Söylemez. *netwire: Functional reactive programming library*. Version 5.0.3. 2018. URL: https://hackage.haskell.org/package/netwire (visited on 02/27/2020).

[SW16] Kensuke Sawada and Takuo Watanabe. "Emfrp: a functional reactive programming language for small-scale embedded systems". In: *Companion Proceedings of the 15th International Conference on Modularity, Málaga, Spain, March 14 - 18, 2016*. Ed. by Lidia Fuentes, Don S. Batory, and Krzysztof Czarnecki. ACM, 2016, pp. 36–44. ISBN: 978-1-4503-4033-5. DOI: 10.1145/2892664.2892670. URL: https://doi.org/10.1145/2892664.2892670.

[Tri20] Ryan Trinkle. *reflex: Higher-order Functional Reactive Programming*. Version 0.7.0.0. 2020. URL: https://hackage.haskell.org/package/reflex (visited on 02/27/2020).

[TS07] Andrew S. Tanenbaum and Maarten van Steen. *Distributed systems - principles and paradigms, 2nd Edition*. Pearson Education, 2007. ISBN: 978-0-13-239227-3.

[Wat18] Tim Watson. *distributed-process-client-server: The Cloud Haskell Application Platform*. Version 0.2.5.1. 2018. URL: https://hackage.haskell.org/package/distributed-process-client-server-0.2.5.1 (visited on 03/21/2020).

[WH00] Zhanyong Wan and Paul Hudak. "Functional Reactive Programming from First Principles". In: *SIGPLAN Not.* 35.5 (May 2000), pp. 242–252. ISSN: 0362-1340. DOI: 10.1145/358438.349331. URL: http://doi.acm.org/10.1145/358438.349331.

[WW17] Well-Typed LLP and Tim Watson. *Cloud Haskell Platform.* 2012-2017. URL: https://haskell-distributed.github.io/documentation.html (visited on 03/21/2020).

Listings

List of Figures

List of Tables

Glossary

Anti-Message Used in Time Warp synchronization to cancel sent states after a rollback.

Arrowized Functional Reactive Programming Functional Reactive Programming focussed on Signal Functions using the Arrow type class.

BearRiver An FRP library implemented with Dunai.

Causality Error Causality errors result from the temporally disordered processing of events in a distributed system [Fuj00, p.51].

Client Side Prediction A consistency maintenance mechanism. It predicts the behavior of entities of a distributed application [Ber01].

Cloud Haskell A distributed computing framework implemented in the Haskell programming language [EBP11].

Command A command is an input captured by a client. It is sent to a server to be processed.

Consistency Ideally, at any time throughout a distributed application, participating users are presented the same state. This is impossible to obtain due to the inherent delay between sending and receiving of a message over a network [DWM06a, p.221].

Consistency Maintenance Mechanism Techniques to create adequate consistency of distributed systems. Optimistic mechanisms, on the one hand, perform calculations during a DIA and assume that no causality errors occur. In the event of an error, optimistic mechanisms have techniques to restore consistency [Fuj00, p.97]. Conservative mechanisms on the other hand, strictly avoid the disordered processing of events. The objective of conservative mechanisms is then to recognize when it is safe to process an event [Fuj00, p.54].

Dead Reckoning A consistency maintenance mechanism. It predicts positions of graphical entities by extrapolating known values [DIS94]. Fujimoto [Fuj00, p.207] lists three DRMs that solve extrapolation in different ways based on the information that is available. Zeroth-order DRM does not extrapolate values. First-order DRM extrapolates with the velocity of entities, while second-order DRM extrapolates with the acceleration.

Dead Reckoning Model There are different types of Dead Reckoning Models, which solve extrapolation of positions in different ways, based on the information that is available.

Delta Time The passed time since the last application of an SF, represented by the type `DTime`.

Distributed FRP Execution of FRP on a distributed system.

Distributed Interactive Application A type of distributed system modeled as an input-output process.

Distributed Paddles The example application demonstrating the library of this project.

Distributed System A distributed system, as defined by Tanenbaum and van Steen, is a set of autonomous, collaborating computers that appears as one consistent system [TS07, p.2].

Dunai A framework for the implementation of FRP based around the notion of Monadic Stream Functions.

Event A discrete Signal.

Feedback Cycle A feedback cycle means that a part of an output of an Arrow is being redirected as an input to the Arrow itself [CE01, p.6].

Frame A single step in the execution of a DIA, where Commands are captured and state is computed. Each frame is associated with a number, the `FrameNr`.

Functional Reactive Animation The first incarnation of FRP focused on the creation of animations.

Functional Reactive Programming A programming paradigm that regards values that change over time as Signals.

Global Control Mechanism Used in Time Warp synchronization to reduce memory usage and to safely perform I/O operations.

Latency The length of the delay, that occurs when a message is propagated from one node in a network to another, is called *network latency* or just *latency* [SKH02, p.88].

Link A link can be set up between Processes in Cloud Haskell. When a Process terminates, an asynchronous exception is thrown in Processes linked to it.

Local Control Mechanism Used in Time Warp synchronization to prevent causality errors.

Message-Passing A type of communication in a distributed system that exchanges messages instead of using shared data structures.

Monadic Stream Function Generalized Signal Functions used in Dunai, that are parametrized with a monadic type.

Monitor Processes in Cloud Haskell can monitor each other. When a process terminates, a message is sent to all processes currently monitoring it.

Process A computation in Cloud Haskell that has access to message-passing mechanisms in the `Process` Monad.

Reactimate Reactimate means execution of FRP through input-output processes.

Serializable To be able to transmit messages over a network, types to be sent must implement the Serializable type class. All types that implement the Binary and Typeable type classes are of type Serializable.

Signal A value that changes with time. The polymorphic type `Signal a` equals functions that map values of time to values of type `a`: `Signal a = Time -> a` [NCP02, p.52].

Signal Function Signal Functions are used in FRP and map Signals to Signals. Represented by the type `SF a b`, where `a` denotes input and `b` denotes output.

Straggler Message Messages with timestamps lower than the current time in Time Warp. Straggler messages lead to rollbacks.

Time Warp A consistency maintenance mechanism. Time Warp allows an application to reset to earlier states to process messages at the time they occurred [Jef85]. This procedure is called a rollback.

Type class of Arrows The type class of Arrows was introduced by Hughes in 2000 and forms a generalization of Monads [Hug00, p. 78].

Typed Channel Typed channels in Cloud Haskell are used to enable type-safe communication by only allowing messages of a specific type to be sent. They consist of a `SendPort` and `ReceivePort`. A `SendPort` allows values to be sent to a `ReceivePort`.

Update An update is the state of an application computed by a server.

Abbreviations

AFRP	Arrowized Functional Reactive Programming
BPMN	Business Process Model Notation
DIA	Distributed Interactive Application
DRM	Dead Reckoning Model
FPS	Frames per Second
Fran	Functional Reactive Animation
FRP	Functional Reactive Programming
GHC	Glasgow Haskell Compiler
LAN	Local Area Network
MB	Megabyte
MSF	Monadic Stream Function
SF	Signal Function

A Additional Listings

Now follow additional listings as supplementary material to the main text.

Listing A.1 shows the unit test `testSendState`. All tests in this project are implemented using the package `tasty-hunit` [Che19]. The type `Assertion` denotes the type of a test. The binary operator `(@?=)` compares its arguments for equality. The function `assertBool` checks if its second argument is true. If one of the functions fails, so does the test. The type of an `Assertion` and mentioned functions stem from `tasty-hunit`[Che19].

LISTING A.1: A sample unit test. Verifies sending of states of a `LocalServer`. Comments explain the test procedure.

```

-- some types used here
data TestMessage = Ping | Pong
  deriving (Generic, Show, Typeable, Eq)
instance Binary TestMessage

type TestUpdate = UpdatePacket TestMessage

-- Test whether writing into the sendVar of a Server
-- actually sends data.
testSendState :: (Node.LocalNode, T.Transport) -> Assertion
testSendState (n, _) = withServer
  (startServerProcess (testConfiguration n) :: IO
      (LocalServer TestMessage TestMessage)
  ) test n
 where
  test server = Node.runProcess n $ do
    (Right sPid) <-
      P.liftIO . atomically . readTMVar $ pidApiServer server

    -- create a channel which will receive states from the server
    (sp1, rp1)   <-
      P.newChan :: P.Process
        (P.SendPort TestUpdate, P.ReceivePort TestUpdate)
    (sp2, rp2) <-
      P.newChan :: P.Process
        (P.SendPort TestUpdate, P.ReceivePort TestUpdate)
    P.linkPort sp1
    P.linkPort sp2

    let sendVar' = (sendVar server)
    let tm1      = UpdatePacket sPid 0 Ping
    -- write a message to sendVar
    writeV sendVar' [(sp1, tm1)]

    -- test: rp1 has tm1, sendVar is empty
    tm1' <- P.receiveChan rp1
    P.liftIO $ tm1' @?= tm1
    isNull <- P.liftIO . atomically $ isEmptyTMVar sendVar'
    P.liftIO $ assertBool "sendVar should be empty" isNull

    -- write multiple messages
    let tm2 = UpdatePacket sPid 1 Pong
    let tm3 = UpdatePacket sPid 2 Pong
    writeV sendVar' [(sp1, tm2), (sp2, tm3)]

    tm2' <- P.receiveChan rp1
    tm3' <- P.receiveChan rp2
    P.liftIO $ tm2' @?= tm2
    P.liftIO $ tm3' @?= tm3
    isNull' <- P.liftIO . atomically $ isEmptyTMVar sendVar'
    P.liftIO $ assertBool "sendVar should be empty" isNull'
```

Eidesstattliche Erklärung

Ich versichere, dass die Masterarbeit mit dem Titel "'Distributed Systems Extensions for the Dunai FRP Library"' nicht anderweitig als Prüfungsleistung verwendet wurde und diese Masterarbeit noch nicht veröffentlicht worden ist. Die hier vorgelegte Masterarbeit habe ich selbstständig und ohne fremde Hilfe abgefasst. Ich habe keine anderen Quellen und Hilfsmittel als die angegebenen benutzt. Diesen Werken wörtlich oder sinngemäß entnommene Stellen habe ich als solche gekennzeichnet.

Leipzig, 11. Mai 2020 Unterschrift