Distributed Reactive Programming for the Web

Alec Dorrington
Curtin University
Perth, Australia
alec.dorrington@protonmail.com

Abstract

Functional reactive programming (FRP) and multi-tier (MT) programming are both promising techniques for designing interactive web applications. FRP abstracts over the flow of *time* in an application, allowing *reactive* variables, which update automatically, to be defined by their relationship to other reactive variables. MT programming semantically combines the client and server portions of an application into a single whole. The combination of these techniques, MT FRP, is just as promising yet seldom used in practice.

A new multi-tier functionally reactive web framework, Chronos, is designed and implemented in Scala. Chronos improves upon existing implementations by adding higher-order reactives, complete glitch freedom, and simultaneous update propagation. Several new techniques are introduced in the process, including the use of time-dependent variables (TDVs), combined push-pull evaluation, and the use of a global deletion threshold for removing old values.

The decisions made and challenges faced in the design and implementation of Chronos are chronicled in the work that follows.

CCS Concepts: • **Software and its engineering** → **Data flow languages**; *Functional languages*; • **Computing methodologies** → Distributed programming languages.

Keywords: functional reactive programming, multi-tier programming

1 Introduction

Many modern computer applications are designed for the web, where content is served by a central server to each user, for each of whom their browser is responsible for rendering this information in a graphical format. This architecture presents many practical problems for the software developer responsible for designing these kinds of applications. Programmers often must toil with client-server message passing, asynchronous event handling, and separate client and server architectures, all while wrangling uncertain or unclear code execution order. A multitude of libraries, frameworks, and programming languages have been designed in attempt to address these challenges.

The following explores a promising approach known as multi-tier functional reactive programming. The idea is to combine two separate techniques, *functional reactive programming* and *multi-tier programming*. Separately, these paradigms have each seen some use in practice, but their combination is rarely seen. While the combined approach may be seldom used, it has seen some modest attention from other authors. Notably, Gavial [1] is a multi-tier functional reactive framework for web applications in the Scala programming language. The following work, however, aims to improve upon existing designs by introducing additional functionality such as higher-order reactives, custom push-pull semantics, and complete glitch freedom. In doing so, *Chronos* is introduced, a proof-of-concept web framework developed to explore and improve upon this new paradigm. It is hoped that further research into this area can drastically reduce development and maintenance costs for future web applications.

2 Background

GUI applications in general, and web applications in particular, are often highly event-driven. In place of a well-ordered and sequential control flow, a web application must respond to frequent and asynchronous user input and network events. A nested hierarchy of callbacks can make a program difficult to reason about and execution order uncertain [2]. This can lead to unforeseen bugs and puzzling race conditions. Reactive programming is a solution that has recently been

gaining in popularity [3]. It is a declarative approach for modelling time-varying values and the interactions therebetween. Multi-tier programming, on the other hand, intends to abstract over asynchronous network communication altogether, where the aim is to reduce or eliminate any semantic gap between client and server by treating both *tiers* as a single application. Both of these techniques are detailed below, separately and then in combination.

2.1 Multi-Tier Programming

Web developers often find themselves using multiple different programming languages within the same broader application. The browser demands the use of *JavaScript* (JS) on the client-side. However, the server imposes no such constraint, allowing the programmer to select the most-fitting or their preferred language and frameworks for at least this half of the application. The client and server each form a separate *tier*. The use of two languages is not necessary, however. For instance, JS can be used on the server-side, too, notably with platforms like *Node.js*. Alternatively, another language can be compiled to JS. Languages like *TypeScript* are designed as better versions of JS, which can be automatically converted to JS for use in the browser. Other languages like Scala and Rust also have JS and WebAssembly compilers available, respectively, the former using ScalaJS. While Scala was originally built for the JVM, it now supports multiple compilation targets, including JS and native.

The client and server portions of a web application can be written in the same language if the language supports compilation to both tiers. The obvious advantage is that mental overhead is reduced from not having to handle two different languages. A more profound advantage is in the possibility of creating *cross-tier* abstractions. That is, certain functions and data structures may be relevant to both tiers of the application. If a multi-target language is used, these only have to be defined once and are automatically compiled into each tier [4].

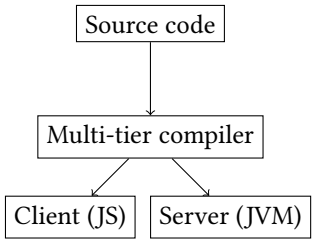


Figure 1. A multi-target compiler is able to produce programs for different environments from the same source code. A multi-*tier* compiler allows these separate environments to talk to one another automatically.

These facts alone do not necessarily make a language multi-tiered. What matters for this classification is the way the boundary between the tiers is handled. A language is not multi-tiered if the client and server portions are treated effectively as separate applications which must communicate explicitly by web protocols. A multi-tiered language is one that can abstract over this glue code, making the message passing implicit [4]. For this reason, Scala by itself is not considered multi-tiered, even though it supports multiple compilation targets. However, it is possible to turn Scala into a multi-tiered language [5, 6] by implementing inter-tier message massing. This allows the developer to conceive of a distributed application as a single whole.

2.2 Functional Reactive Programming

Reactive programming (RP), broadly conceived, is a programming paradigm by which change is automatically propagated throughout the application. The programmer need not concern himself with *when* the state is updated, but only *how* it is updated, which is why it is commonly said that reactive programming abstracts over the flow of *time* in an application. Reactive programming, in this sense, is already widely used in web application development, notably with frameworks like React and Angular. *Functional* reactive programming [7, 8] (FRP), however, refers more specifically to the use of reactive techniques in a functional context. That is, functional programs are those which are described purely by compositions of *referentially transparent* expressions, without the unprincipled use of side-effects.

More concretely, reactive programming establishes dependencies between different (time-varying) *reactive* variables (reactives). The value of a reactive often depends on some number of other reactives via a transformation that describes the value of that reactive in terms of those of its parents. A directed-acyclic graph (DAG) known as the *dataflow graph* is maintained, which represents this set of dependencies, throughout which updates are automatically propagated when new events are triggered. Acyclicity is required because otherwise, non-converging relations could cause nonterminating propagation.

As an example, consider a simple reactive system with two values x and y. Let y be defined such that $y = x + 1$. In this case, y *depends on* x, and x is said to be y's parent. When a new value is assigned to x (assume by some exogenous cause), y will be automatically updated, too, by the given relation (incrementing by one, in this case). Thus, the value of y will *always* be one greater than the value of x, no matter what changes the system undergoes.

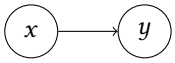
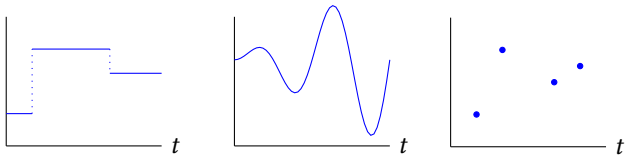


Figure 2. Example dataflow graph for the simple case with two reactive values x and y, where y depends on x.

2.2.1 Events and Signals.

It is common to separate reactives into two types; events and signals [7]. Events are

discrete and instantaneous. They are often triggered in response to things happening outside the reactive system (the press of a button, the receipt of a network packet, etc.). Signals, on the other hand, are *sustained* values; they persist over time. Signals change their value in response to other signals they depend on doing the same, or, ultimately, in response to any events which may occur. Events are what usually cause changes to a system's state. Some reactive systems [9] are said to be *continuous*, meaning they allow for signals to change not only in response to events but also continuously over time.



(a) Discrete signal *(b) Continuous signal* *(c) Event stream*

Figure 3. Visual comparison of the different types of reactive values and their relationship to time. Signals persist over time, but events don't.

2.2.2 Reactive Combinators. Events and signals are composed to form entire reactive programs [10]. Different FRP frameworks provide different sets of combinators for this purpose. Operators like *map*, *filter*, *zip* and sometimes *flatMap* are provided so that new reactives may be created as combinations of their parents. The use of combinators to form new reactives from existing ones builds a dataflow graph *implicitly*. Support for *flatMap* is less common because it allows for and requires *dynamic* dataflow graphs, which may change their topology throughout the execution of the program [11]. This has its benefits but makes the implementation more difficult [12], and is discussed further in the section on *higher-order reactives*.

2.2.3 Push and Pull. Reactive programs are often categorised as being either *push-based* or *pull-based*. In a push-based system, when a reactive is updated, its dependents are updated immediately in turn, and so forth; evaluation is said to be eager. In a pull-based system, reactives aren't updated until they are needed. Querying the value of a reactive will have it query the values of its parents, and so forth; evaluation is said to be lazy. Push-based evaluation has the advantage that propagation through certain subgraphs can be skipped if they are invariant under certain changes, and that no extra delay is imposed by a finite polling frequency [13]. Pull-based evaluation, on the other hand, is more compatible with *continuous* signals, or just signals which frequently change, where the required sampling frequency is much less than the source frequency. Some systems combine push and pull evaluation to obtain maximum advantage.

2.2.4 Higher-Order Reactives. Some reactive frameworks provide support for *dynamic* dataflow graphs, which change their structure over time. For instance, Yampa [11], an FRP library for Haskell, provides a *switch* combinator, which allows for reactives to change their parent. The idea is to use the value of a regular, static parent to select another parent dynamically. The switching reactive takes the value of the dynamic parent as its own. When the value of the static parent changes, so does the dynamic dependency and thus the structure of the dataflow graph. The switching reactive is said to be *higher-order*. To motivate this use case, consider a GUI with tabs, where the contents of each tab are described as reactive values. The content shown is *switched* in response to the tabs being clicked. Also consider a distributed case where devices are free to join and leave a network at any time; a dataflow graph that can change over time is needed to describe this behaviour [13]. This switching combinator allows a suitable *flatMap* operation to be defined, thus making signals semantically *monadic* [13]. Implementations often don't offer this functionality, however, since it makes state propagation more difficult. For instance, Gavial doesn't support higher-order reactives at all.

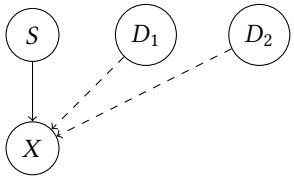


Figure 4. An example of a dynamic dataflow graph using X as a switch. X decides which reactive to copy, D_1 or D_2, based on the value of its static parent S.

2.2.5 Glitch Avoidance. Consider a case with four reactives, a, b, c, d, where a depends on nothing, b and c both depend only on a, and d depends on both b and c. A diamond structure is formed. Assume the value of a is modified. In situations such as this, it is important to consider in which order the dependent reactives are updated. Obviously, a is updated first. From here, it doesn't matter in which order b and c are updated, and this could even be done in parallel. If both b and c are updated, then d can be updated without problem. However, if b completes its update before c and triggers an update to d in turn, then this can cause some undesirable behaviour. The new value of b and the old value of c might be *mutually inconsistent*; each by itself may be perfectly legitimate while their combination describes an invalid state. This of course can lead to undefined behaviour in the determination of d. Situations such as this are known as *glitches* [7] in the reactive programming literature. The effect is only temporary since when c is finally updated, d will update again to reflect the new state. However, even a temporary anomaly may be undesirable. Consider the case

where a is some transaction event, and b and c each represent the bank balance of two different users. One might expect the total amount of funds held by all users to be invariant under internal transactions like this. If d is this total, the stated invariance property fails to hold in a system with possible glitches.

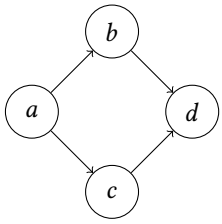


Figure 5. The simplest case of a dataflow graph with possible glitches. If the nodes update in the order a, b, d and then c, the node d could temporarily be in an inconsistent state.

The solution is to ensure each reactive only updates after *all* of its parents have. One option is to sort all reactives by minimum distance from each possible event source, and then propagate updates in this order. Reactives at the same depth can be processed in parallel. This works well for the simple single-tiered, static-graph case. However, with an application spanning multiple tiers, the global topology of the dataflow graph won't be entirely known to any given tier. This is made especially problematic when the topology is able to change dynamically. This method is also unable to take maximum advantage of parallelism in that reactives at greater depths must wait on all reactives at lower depths, even non-ancestors.

A better solution is to, upon the update of any reactive, send a message to each of its children, informing them that the update has occurred. Each reactive maintains a record of which parents have and have not updated, which is modified upon receipt of such a message. The child will update itself if and only if all of its parents have already updated. This approach is largely decentralised and requires no global knowledge of network topology. A reactive need only know of its direct parents and children. This is precisely the approach taken by *SID-UP* [14], a *distributed* reactive programming propagation algorithm which guarantees complete glitch-freedom.

However, a significant downside of SID-UP is that only a single propagation (from a single event source) may take place at a time. That is, a global lock on propagation must be maintained across all tiers. For the sake of responsiveness and efficiency, this can be undesirable, especially for events that may not have much to do with one another. *QPROP* [15] improves upon the idea by allowing simultaneous propagation from different sources. Instead of storing only the latest value, each reactive maintains a value *history* with associated timestamps, from which non-glitched parent-value combinations can be deduced. That is, the value history of a given reactive is determined by a Cartesian product of its parent's value histories, filtered so that only those values which existed in overlapping time periods are considered.

2.3 Multi-Tier FRP

Multi-tier functional reactive programming is the combination of these two techniques, MT and FRP, with a few additional stipulations. That is, a language or framework [1, 16, 17, 18] can be said to support MT FRP if all of the following conditions are met:

- `Implicit tier boundary`: Values can be sent between tiers without reference to the underlying communication channel (multi-tier programming).
- `Reactive dependencies`: Time-varying values can be defined in relation to other such values, with those relationships being automatically maintained (reactive programming).
- `Referential transparency`: All expressions are referentially transparent without arbitrary use of side-effects (functional programming).
- `Distributed reactives`: Reactive values are allowed to exist on any tier.
- `Multi-tier reactives`: Reactives on different tiers are allowed to depend on each other.

The first three conditions are already implied by the use of MT and FRP. The final two are what make the combined approach special. If reactive dependencies are allowed to span multiple tiers, reactive update propagation becomes the method by which client-server communication occurs. This allows for the fullest integration of reactive programming into the client-server model.

2.3.1 Tier-Crossing Operations. A multi-tier framework will abstract over network communication, but this doesn't mean the programmer has no control over *where* network communication happens. Every reactive value is tied to a specific tier, there are *client*-specific values and *server*-specific values. For a reactive to depend on values that exist on other tiers, *tier-crossing* operations are used [10]. For example, *toServer* (*toClient*) can be used to convert a client (server) reactive into an equivalent server (client) one. Value updates are automatically sent across the network, just as is the case for regular reactive dependencies (except with a network in between). Reactives that exist on the tier boundary thus have their value duplicated across both tiers [19].

3 Objectives

The objective of this project is to design and implement a new multi-tier functionally reactive web framework, called *Chronos*. Since there are very few such frameworks, it is hoped that this project will help further development in this area. In addition to meeting the five conditions for MT FRP listed previously, the new framework must expand upon

existing implementations by combining the following set of features. The incorporation of these features introduces a new set of challenges that must be faced, and the combination of which should produce a web framework that is sufficiently expressive so as to be able to build a wide variety of potential web applications.

- `Total glitch freedom`: Reactives must update in a perfectly sound order without the possibility of glitches. Glitch avoidance shouldn't be the responsibility of the application developer.
- `Higher-order reactives`: In real-world applications, dependency structures can change over time. A reactive framework must allow for the possibility by permitting certain reactives to dynamically change their parents
- `Asynchrony`: Multiple propagations must be able to take place in *parallel*, whether they be from the same or different event sources. This is necessary for applications to take advantage of multi-core processors and the distributed nature of web applications.
- `Scalability`: An application must be able to scale in accordance with the number of connected devices. This necessitates that each device operates *autonomously* so as not to overload a central coordinator.

While facilities to work both with GUIs and databases are essential for successful web development, these are both outside the scope of this project. As an FRP framework, Chronos needs to handle event and value update semantics, and as an MT framework, it needs to handle inter-tier message passing. GUI interactivity and database access can be done by other libraries which are either separate to or built on top of Chronos. That is, the ultimate intention is for Chronos to act as a sort of base layer on which further technologies can be built or integrated. For the time being, however, Chronos is mainly intended as a proof-of-concept. For more on this, see the section on *Future Work*.

What follows is an outline of each of the design decisions made by Chronos, an exploration of the challenges encountered by trying to combine these features, and detailed implementation details which go over solutions to these problems.

3.1 Distributed Glitch Freedom

Measuring the degree of glitch freedom is less straightforward in the distributed case. Four cases are identified, as follows in order of increasing safety:

- `Glitch prone`: No effort is made to prevent glitches, which may occur at any time.
- `Local glitch freedom`: Within each tier, glitch freedom is assured. However, propagation *between* tiers may involve glitches.
- `Tiered glitch freedom` [1]: No network communication is done until all local propagation is completed, then foreign updates are sent all at once. A propagation which only crosses a tier boundary in one direction is guaranteed to be glitch-free. A propagation that crosses the same tier boundary in multiple directions may still experience glitches.
- `Total glitch freedom` [21, 20]: All glitches are impossible.

Gavial introduces and implements the idea of tiered glitch freedom. However, this is considered insufficient for this project.

4 Method

4.1 Software

Chronos is created in and for the *Scala* programming language, because it supports (a) functional programming and (b) multiple compilation targets (client and server). Since the reactive programming model makes use of category-theoretic concepts (signals and events are both monads), the Scala *Cats* (short for categories) library is also used [22]. This provides a consistent interface for operating on reactives, given only a few simple type class implementations. An actor-based approach [23] is used for update propagation. Each reactive value has a corresponding actor which maintains its state. The *Akka Actors* framework [24] is used for this purpose. Sending updates between client and server is done via WebSocket for bidirectional communication, for which *Akka HTTP* is used, a popular HTTP library in Scala. Finally, *Circe* is used for JSON serialisation and deserialization of any objects sent across the network boundary. This is done by type class, with macros for case classes and sealed traits, making it a largely seamless developer experience.

4.2 Design

4.2.1 Primitives. Each reactive system in Chronos is built up from a small collection of basic primitives:

- `Events` and `Signals`: Both of which are defined in ways consistent with the broader FRP literature, as outlined previously. All signals in Chronos are discrete. In Chronos, events and signals are both *monads*.
- `Sources`: Source events are a special kind of event. Source events ultimately cause all changes to the state of the reactive system. These act as an interface between the rest of the application and the reactive system. User input actions (mouse/keyboard), for instance, are converted into reactive events via sources.
- `Sinks`: Sinks enable effectful responses to reactive events. That is, sinks act as an interface between the reactive system and the rest of the application, in the other direction. Sinks are what allow the reactive system to actually *do stuff*. Sinks are also *contravariant functors*.
- `Systems`: The simplest system is the pairing of an event and a sink. This would configure that particular

sink to respond to that particular event stream. Systems can be *combined* to form more complex multiple-sink systems. Accordingly, systems can be thought of as *monoids*.

Chronos has support for *dynamic* topologies, meaning the set of events, signals, and sources in a system can change over time. However, the set of sinks will always remain unchanged. A system is uniquely identified by the set of event-sink pairs it contains. The server and client will each run different portions of the reactive network, but they each still have *identical* system instances (meaning the same set of sinks).

4.2.2 Combinators.

- `map`: Both events and signals have a *map* operator. $R.map(f)$ produces a new reactive with R as its only parent, according to the relation f.
- `zip`: Events and signals both also have a *zip* combinator for combining values from multiple parents. This operation is very natural for signals, but can be defined for events too by tupling a current event from one parent with the most recent past event from the other parent.
- `flatMap`: To allow support for higher-order reactive programming, events and signals are given *flatMap* combinators which act as topology switches. For a signal switch, depending on the value of a single static parent, select which other reactive to *copy* until the static parent's value changes. For an event switch, each time the static parent is triggered, select which other event stream to mirror.
- `filter`: According to some predicate, prevent certain event instances from being received downstream. Only events (not signals) can be *filtered*.
- `combine`: Merge multiple event streams, so that event instances from either are received downstream. Again, this is only available for events.
- `scan`: Incrementally generate a new event stream from a given one. An event is produced downstream when one is received from upstream, given as a function of this upstream event and the previous downstream one.
- `hold`: Convert an event into a signal. There needs to be some way for events to affect the values of signals, and this is it. The value of the output signal is equal to the *most recent* event instance to have occurred in that stream.
- `fold`: A combination of a *scan* and a *hold*. Incrementally generate a signal value based on all of the event instances leading up to that point. *Hold* and *fold* are also both used by Gavial in the same way.
- `emit`: Given some event E and signal S, produce a new event which triggers with the current value of S whenever E triggers. The value of E is unused. This is similar to the *tag* combinator from Yampa, which attaches to an event the value of some Signal at that time.
- `changed`: Produce an event that triggers whenever the given signal's value changes.
- `sample`: Set the update frequency of a signal. If some signal updates at a much higher (or lower) frequency than what is required, then *sample* will limit the update rate to that of the frequency specified. This is included for performance reasons, since it may not be desirable to be constantly sending updates over the network when it isn't necessary for a value to always be completely up-to-date.

The inclusion of *map* operations makes events and signals *functors*. With *zip* combinators, and the existence of *pure* constructors (signals of constant, unchanging value and events which only trigger once, at genesis), they are both *applicative* functors. Most FRP implementations will stop here, but events and signals are also *monads* if a *flatMap* combinator is allowed, as is the case in Chronos. Further, events but not signals are also *monoids*, given the *combine* operation for merging event streams, where the identity is an empty event stream. While *Cats* normally will automatically derive applicative-style combinators from monadic ones, a separate implementation for these is still provided so that the performance advantages of a *static* topology can be realised wherever possible.

To put it all together with an example, consider a case where one needs to detect the press of an on-screen button. Given an existing *MouseClick* event, this can be *filtered* twice, based on (a) the position of the cursor, to make sure the cursor is inside the button, and (b) the physical button pressed, to make sure the on-screen element is *left*-clicked. If the position of the on-screen button is itself a reactive signal, the *emit* combinator can be used in (a) to compare the cursor position to the button position. This can then be *mapped* to change the *MouseClick* event into a *ButtonClick* event. The above could all be handled by a GUI library, exposing only the final *ButtonClick* event to the application programmer. To count the number of times the button is pressed, this *ButtonClick* event could be *folded* to produce a signal which increments itself when a click is registered.

4.2.3 Three Tier Model.

While the client-server model for web applications has only two tiers, it is possible and often convenient to introduce additional *logical* tiers. Consider: the server and client tiers exist in a one-to-many relationship. Say there exists an event on the client, on which a *toServer* operator is used. Should this new server event stream contain events from just one client, or from all clients?

When going from the 'many' to the 'one' side of the relationship like this, it would make sense to *merge* the event streams from each client, tagging each event with a unique identifier for the client from which it originated. Going the other way, events should specify whether they want to be

Table 1. Chronos combinators for $Event[A, T]$. A is the type of value *channeled* by the Event, and T is the tier to which the Event belongs.

Combinator	Parameters	Result
filter	p: A => Boolean	Event[A, T]
map[B]	f: A => B	Event[B, T]
zip[B]	s: Signal[B]	Event[(A, B), T]
flatMap[B]	f: A => Event[B]	Event[B, T]
combine[B]	Event[B, T]	Event[A | B, T]
emit[B]	s: Signal[B]	Event[B]
scan[B]	initial: B, f: (B, A) => B	Event[B, T]
hold	intial: A	Signal[A, T]
fold[B]	initial: B, f: (B, A) => B	Signal[B, T]

Table 2. Chronos combinators for $Signal[A, T]$. A is the type of value *held* by the Signal, and T is the tier to which the Signal belongs.

Combinator	Parameters	Result
map[B]	f: A => B	Signal[B, T]
zip[B]	s: Signal[B]	Signal[(A, B), T]
flatMap[B]	f: A => Signal[B]	Signal[B, T]
changed	N/A	Event[A, T]
sample	frequency: Int	Signal[A]

Table 3. Chronos combinators for $Sink[A, T]$. A is the type of value *received* by the Sink, and T is the tier to which the Sink belongs.

Combinator	Parameters	Result
contramap[B]	f: B => A	Sink[B, T]
observe	e: Event[A, T]	System

sent to a *specific* client or broadcast to *all* clients. Signals should work similarly to events, except each boundary value on the server-side is actually a map from clients to values.

However, this model is inconvenient when the behaviour required from the server by each client is mainly independent. Consider, an event is sent from the client requiring the server to perform a database query and return the result. The reactives on the server-side of the boundary must explicitly track the ID of the client who made the request in order to make the response to the correct device. If the tier boundary were one-to-one instead of one-to-many, this problem would be avoided. One can instead use a framework that tracks this session information *implicitly*.

The full generality of the first case is required when an application involves any client-to-client interaction. However, this increased power makes writing simple request-response

Table 4. Chronos tier-crossing operations for client and session Events in the form $Event[A, T]$ where $T \in \{Client, Session\}$.

Operator	Result
toClient	Event[A, Client]
toSession	Event[A, Session]
toApplication	Event[(SessionId, A), Session]

Table 5. Chronos tier-crossing operations for application Events in the form $Event[(Set[SessionId], A), Application]$.

Operator	Result
toClient	Event[A, Client]
toSession	Event[A, Session]

style applications more difficult. However, it is possible to get the best of both worlds. For instance, Gavial uses a three-tiered approach, with two separate logical tiers both corresponding to the physical server. The *application* tier exists in a one-to-many relationship with all of the clients, and the *session* tier exists in a one-to-one relationship with a *specific* client. For each of these tiers, Gavial includes a tier-crossing operation; *toClient*, *toSession* and *toApplication*. Thus, the problem is solved. Chronos uses the same three-tier model and tier-crossing operations as introduced by Gavial.

Events and Signals are both typed according to (a) the type of value they contain and (b) the tier to which they belong. Except for with the use of tier-crossing operations, only events/signals which belong to the same tier can be combined.

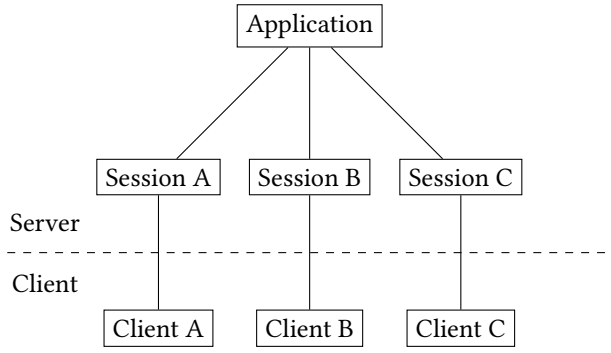


Figure 6. The three-tier model implemented by Gavial. Each session tier on the server corresponds to exactly one client. The application tier allows for interdependence between the different sessions.

Table 6. Chronos tier-crossing operations for client and session Signals in the form $Signal[A, T]$ where $T \in \{Client, Session\}$.

Operator	Result
toClient	Signal[A, Session]
toSession	Signal[A, Session]
toApplication	Signal[Map[SessionId, A], Session]

Table 7. Chronos tier-crossing operations for application Signals in the form $Signal[Map[SessionId, A], Application]$.

Operator	Result
toClient	Signal[A, Client]
toSession	Signal[A, Session]

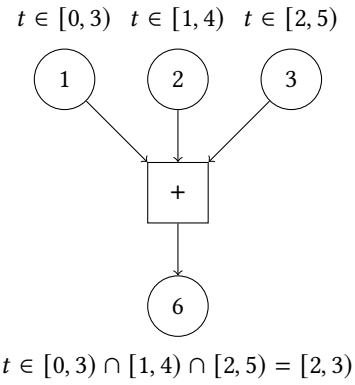


Figure 7. For a propagation to be sound (without possibility of glitches), the parent values must be defined in overlapping time periods.

4.3 Implementation

4.3.1 Simultaneous Propagation. Traditional glitch avoidance methods like SID-UP avoid glitches by only allowing a reactive to update once *all* of its parents have. However, if a second source event is triggered while the first one is still propagating, the concepts 'updated' and 'not updated' are not well defined. Does updated mean with respect to the first event, or the second, or both? For this reason, SID-UP employs a global propagation lock to prevent a new propagation from starting while there is already one ongoing; the second event must wait for the first to finish. This is very bad for scalability, especially when the two source events don't have much to do with one another.

To solve this problem, the current value of each signal can be tagged with a start time and end time. The start time is the time of the source event that first caused the reactive to take this value. The end time is the time beyond which the reactive no longer knows whether or not it is up-to-date. A stage in the propagation consists of taking the values of each of the parents of a reactive R, combining them in some way, and setting the result to be the new value of R. Such a propagation is considered sound if the intersection of the time periods for each of the parent values taken is non-empty. This intersection becomes the new period for R itself.

The reason for this is as follows: if the intersection is non-empty, then there exists some time t at which all of the parent values are completely up-to-date, but also not from the future. This *must be* a mutually-compatible set of values. Consider, a glitch describes a situation where a reactive draws parent values from *different* times that aren't compatible. This condition generalises the notion of 'don't update until all of your parents have' to the case of simultaneous propagation. Further, each possible t describes a case where the combination is compatible, which is why the intersection gives the range of eligibility for R.

But what if this condition is not met, and the parent's values have non-overlapping time periods? The solution is for each parent to store not just the most recent value but a *history* of values. When the values are given to the child during propagation, only the *difference* between the current and previously-given value history is provided.

4.3.2 Time-Dependent Values. Both signals and events have values that are time-dependent. The value of an event is a mapping from times to values. The value of a signal is a mapping from time *periods* to values. To handle these kinds of values, a new abstract data type (ADT) is introduced; the *time-dependent value* (TDV). TDVs store not just a single current value, but a *history* of values across time. This is implemented as a binary search tree (BST) of time-value tuples, sorted in chronological order. There are two different kinds of TDV; those indexed by time and those indexed by time *range*. Time ranges are not allowed to overlap. These are each necessary for supporting events and signals, respectively. When it is necessary to differentiate, these are known as event-TDVs and signal-TDVs.

TDVs are equipped with several elementary operations:

- `combine`: Merge the time-value pairs from both. This allows a TDV with old values to be combined with a TDV with new values, resulting in a more complete history. For signal-TDVs, ranges must be non-overlapping or otherwise mutually compatible.
- `map`: Modify the values without affecting the times.
- `zip`: Produce a new TDV containing entries for every *pair* of values from two TDVs to exist in overlapping time periods.
- `flatMap`: For each time period in the TDV, specify another TDV for which the values in this same period should be taken.

- `crop`: Limit a TDV to a specified range. This is used (a) to avoid sending redundant data and (b) as a subprocedure for the above *flatMap* operation.

Note that TDVs are both monoids and monads, due to *combine* and *flatMap* respectively. For event-TDVs, *time period* in the above should be interpreted as the gap between two consecutive events.

The usefulness of these operations corresponds directly to the need to perform analogous operations on the reactives themselves. That is, for instance, when two signals are *zipped*, the update procedure performed by the resulting signal is simply to zip the respective TDVs from each parent. Thus, the requirement for TDVs to be monads instead of just applicatives corresponds directly to the fact that Chronos has support for higher-order reactives.

For some signal-TDV X, a given time t may or may not be enclosed by a range $r = (t_0, t_1)$ from a range-value pair (r, v) in X. If this is so, v is said to be the value of X at time t. Note that a TDV need not be defined at every time t. In practice, a reactive will only track a small portion of the value history at any given time, making *partial*-TDVs the predominant use case.

A boolean-valued signal-TDV is called a *temporal mask*. A temporal mask associates with each time t a value of either *true* (T), *false* (F) or *unknown* (U). The standard set operations (*intersection*, *union*, *complement*) are defined for temporal masks, working in accordance with Kleene three-valued logic. That is, $T \vee U = T, F \vee U = U, T \wedge U = U, F \wedge U = F, \neg U = U$. If $M(t) \in \{T, F, U\}$ indicates whether mask M includes the time t, to compose masks by *intersection*, *union* or *complement* is to update the inclusion status of each t according to (3-valued) logical *and*, *or* and *negation* respectively.

Each event-TDV E comes equipped with a temporal mask denoted $mask(E)$ which tracks the time period(s) for which this TDV actually has up-to-date information. This is necessary from differentiating between a propagation delay and a situation in which there really are no new updates. Signal-TDVs don't need explicit temporal masks because they can be automatically derived from the union of the time ranges present in the TDV.

Several diagrams are included to illustrate how TDVs compose.

4.3.3 Reactive State.

Each reactive has a corresponding Akka actor. The state of each actor is a 5-tuple (C, P, V, U, W):

- `Children (C)`: stores the *children* of the reactive, as a mapping from reactives to temporal masks. That is, since Chronos supports dynamic topologies, the set of children is not constant across time. Each mask tracks the time period(s) in which that parent-child relationship holds.
- `Parents (P)`: stores the *parents* of the reactive, also using temporal masks.

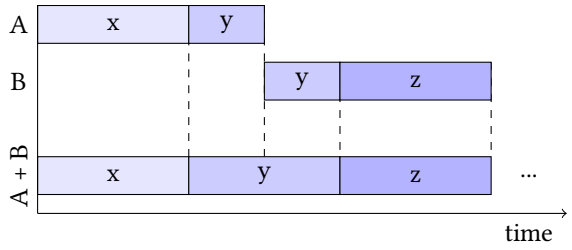


Figure 8. To *add* or *combine* two time-dependent values is to combine their information about which values existed at which times. This is useful for state propagation where old information needs to be combined with new information.

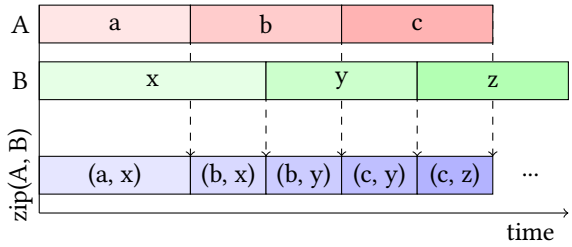


Figure 9. To *zip* two time-dependent values is to form a new TDV featuring every pair of values which existed in overlapping time periods. This operation is necessary for propagating values in a case where one reactive has multiple parents whose values need to be combined.

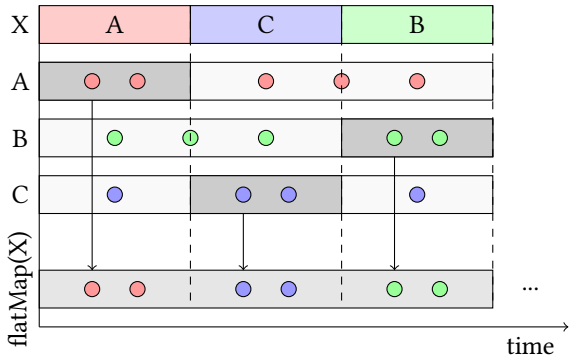


Figure 10. To *flatMap* a time-dependent value is to, for each time period, adopt the values of another TDV in this period. This operation is a generalisation of the zipping operation shown above, and is necessary for performing switching actions where the parent of a reactive can change over time.

- `Values (V)`: is a TDV describing the reactive's own value history.
- `Parent Values (U)`: stores the values histories of each of the parents of the reactive, as a mapping from parents to TDVs.
- `Wait Mask (W)`: is a temporal mask used purely for pull-based evaluation (see below), and indicates the

time period(s) 'currently being pulled' by this reactive from each of its parents.

Note that V mirrors the parent's respective U values, and P mirrors the parent's respective C values. This information is necessarily duplicated for an actor to be able to contextualise its place in the dataflow graph.

The actors communicate using a small set of messages:

- `Subscribe(c, m)`: Inform another actor that it is now a parent of this reactive. Used upon initialisation for static parents, and when the topology is changed for dynamic parents. c identifies the actor which sent the *Subscribe* message. m is a temporal mask indicating the duration of the subscription.
- `Push(p, x)`: Provide a child with more recent values for this reactive. p identifies the parent which sent the *Push* message. x is a TDV containing any relevant upadates.
- `Pull(w)`: Ask a parent to provide updates. w is a temporal mask indicating the time period(s) being requested.

Akka guarantees that each actor will only be responding to at most one message at a time, thus avoiding any possibility of race conditions.

4.3.4 Initialisation and Subscription.

Initially, the reactive system only creates an actor for each of the *sinks*. When an actor is initialised, it sends a *Subscribe* message to each of its static parents. For the mask, a range from the current time into the indefinite future (positive infinity) is used. At any point, if an actor tries to send a message to another reactive which doesn't have an actor, a new actor is created automatically. When a Subscribe message is received, a new mapping in C is created, $c \mapsto m$. If the entry $C(c)$ for c already exists, the value is overridden. An actor can similarly *unsubscribe* at a later time by sending another *Subscribe* message with a different mask.

4.3.5 Push, Push and Polling.

Push and pull based propagation both have to perform some polling. This is perhaps more obvious in the latter case. A sink which wants a more recent value must explicitly ask for it. If it wants to continuously receive new values, it has to poll for them. What is less clear is that push-based propagation requires polling too, just of a different form.

Consider a case where there are two signals, one on the client which changes very often (e.g. cursor position), and one on the server which changes very rarely (e.g. the number of users to have signed up for the site, rounded down to the nearest million). Lets say some visual element depends on both of these values (e.g. a piece of text that reads: "Number of users: X million, cursor position: Y."). Assuming standard push-based evaluation, this program wouldn't work very well with the model of glitch avoidance employed here. The text wouldn't respond properly to the cursor's movement because most of the time there isn't a new value coming from the user count signal. If this were to work, the user count signal would need to periodically send out updates to its children to inform them that the value *hasn't* changed.

This problem is often solved by tracking, for each reactive, the set of source events from which that reactive is reachable. Then, for each parent, only wait for it to update if its actually reachable from the source of the event currently being propagated, otherwise its old value is fine to use. This approach, however, is greatly complicated by the addition of both simultaneous propagation and higher-order reactives. Maybe it currently looks like a reactive doesn't descend from a particular source event, but upstream the topology just switched to introduce such a dependency. In short, pull-based evaluation requires polling to see if anything has changed, and push-based evaluation requires polling make sure *nothing* has changed.

Thus, Chronos introduces a new form of propagation, a hybrid push-pull model. Reactives will always *push* their new values when they are changed. However, when one parent is changed, a reactive will *pull* new values from its other not-yet-updated parents to match. Note that this is not to be confused with [13], since that hybrid model involves using only push-based evaluations for discrete signals and pull-based evaluation for continuous signals.

4.3.6 Push Phase.

Push messages are sent in any of three cases:

1. By a source event, when an update is triggered by something outside the reactive system.
2. By any reactive, to all children, after the reactive updates its own value for V.
3. Whenever a new *Subscribe* message is received, but only to the reactive who sent the request.

The first two cases define a push-based evaluation scheme. The final case is to ensure new children are initialised with up-to-date information.

However, *Push* messages don't just send the entire value history V. This would be inefficient as it would involve sending a lot of repeat data. All that needs to be sent is the *difference* between the current value and what has already been sent. The update x sent to each child c is thus cropped in accordance with the subscription mask $C(c)$ for that child. That is, values are only included from times when the subscription is active:

$$x = crop(V, C(c)) \tag{1}$$

If x is empty, no message is sent. Otherwise, after any *Push* message is sent to a child, the value of the subscription mask for that child is then updated so as to subtract out the mask of the update x, in order to prevent this correspondence from being redundantly repeated:

$$C(c) \leftarrow C(c) \setminus mask(x) \tag{2}$$

When a *Push* message is received from a parent p, the actor updates U with the new parent values x:

$$U(p) \leftarrow U(p) + x \tag{3}$$

Furthermore, the receiving reactive also modifies its parent mask to no longer require older values:

$$P(p) \leftarrow P(p) \setminus mask(x) \tag{4}$$

Using U, the actor tries to update V. If V is modified, more *Push* messages are sent to each of this actor's children, as per case 2 above.

4.3.7 Pull Phase. Whenever an actor receives a *Push* message from a parent featuring an updated value history, it may need correspondingly newer values from its other parents too in order to continue propagation. At this stage, it isn't known whether the other parents are actually unchanged or whether the upstream propagation is still ongoing. It is here that pull-based querying is combined with push-based propagation.

When a *Push* message is received, *Pull* messages are sent to each parent to get them to relinquish newer values. For each of these parents, this would involve sending *Pull* messages to each of their parents in turn, continuing all the way up to the source events. If a source event receives a *Pull* message, it will respond immediately with a *Push* message to inform its children that there are no changes in the specified period(s). This will trigger another forward, push-based propagation.

For cases with diamond-structured dataflow graphs, this would be inefficient due to repeated propagation through certain subgraphs. For cases where there is actually a forward propagation inbound that just hasn't arrived yet, this is also redundant. For this reason, *Pull* messages must specify by w for which time period(s) exactly they are requesting new values, in the form of a temporal mask. That way, an actor can decide *not* to forward *Pull* messages onward if they describe a period that is already evaluated. Thus, when a backwards propagation meets a forwards propagation which renders it redundant, it is annihilated. Furthermore, each actor stores *for which* time period(s) *Pull* messages have been sent to its parents so that these requests are never repeated. This is the purpose of the actor's W mask.

For some parent p, the subscription mask is denoted $P(p)$ and the mask of that parent's value history is $mask(U(p))$. The intersection $P(p) \cap mask(U(p))$ describes the period(s) for which that parent's values are both required and known. The union of these period(s) for all parents p gives the period(s) for which at least some values are required and known. Subtracting off $mask(V)$ removes the period(s) which are up-to-date, just giving those which this reactive is *still waiting for*. This quantity, labelled w, describes the *time periods* for which a reactive R needs updated value histories, if it is to keep up with its most-recently-updated parent:

$$w = \bigcup_{p \in parents(R)} P(p) \cap mask(U(p)) \setminus mask(V) \tag{5}$$

Whenever an updated value history is received from one parent's *Push* message, a reactive sends out *Pull* messages to *each* of its parents p, using the value of $w \cap P(p)$ for the mask, provided $w \cap P(p) \neq \emptyset$, otherwise no new values are needed from this parent.

When a *Pull* message is received for the time period w_0, an actor will:

1. For each of its parents p, send them each a *Pull* message with $w = w_0 \cap P(p) \setminus W$, provided $w \neq \emptyset$.
2. Update the value of W according to $W \leftarrow W \cup w_0$ to prevent this propagation from repeating.

4.3.8 Sampling Frequency. To sample a signal at a frequency other than that at which it naturally updates, a *sample* operation is defined for signal-TDVs. To sample a TDV at some frequency f, discretise time into periods of length f. The value of the original signal at time $\lambda f, \lambda \in \mathbb{N}$ is taken to be the value of the sampling signal throughout the entire range $(\lambda f, (\lambda + 1)f)$. For each value-range pair $((t_0, t_1), v) \in V$, t_0 and t_1 need to each be rounded up to the next multiple of f. Any zero-length ranges can then be deleted, these indicating changes at too great a frequency. Rounding up the end time t_1 also has the effect of *forward-dating* the expiry of each value. This will allow other downstream signals to continue using the old value without checking if it is still up-to-date until the point when the next interval is reached.

The addition of sampling intentionally reintroduces glitches into the model. However, this gives the application developer the best of both worlds; the ability to decide which parts of the application need to be glitch-free and which would benefit more from additional performance. This method of sampling integrates seamlessly into the propagation model described above.

4.3.9 Event Ordering. It is a requirement that all source events are part of a strict global ordering. That is to say, for any two given source events, it must be known which came first. For this reason, timestamps are attached to each source event. However, this isn't enough, since there's always the possibility that two events from different sources receive the same timestamp by chance. Thus, appended to each timestamp is a 64-bit random number. This addition only affects cases that would otherwise be collisions and reduces the probability of an actual collision to effectively zero. This procedure is acceptable because when events are triggered at very similar times, it matters less what the order of the events actually was and more that there is *some* consistent order, no matter what that order is. Any changes to the reactive system which occur downstream as a result of a source event also adopt the same timestamp as the source.

4.3.10 Clock Synchronisation. The proper functioning of the algorithm requires relatively well-synchronised clocks. While out-of-sync clocks won't cause glitches, they can cause some efficiency and reliability issues. Since source events are timestamped according to the time on the local device, if one device has a fast clock, events originating from that device may fail to be propagated in a timely fashion. This is because nodes require updates to be processed in chronological order to prevent glitches. Events from other devices will take precedent, even if in actuality, they occurred later. The propagation delay may be as large as the clock offset. When a WebSocket connection is initialised, a standard clock synchronisation procedure is employed. One device asks the other to send back the time, and half the RTT is compared to the difference between the local and foreign times to obtain the time offset for the other device.

4.3.11 Deletion Threshold. The solution, as it has been presented thus far, would be horribly space inefficient. Every single value ever held by any reactive would be stored in perpetuity. Consider an event triggered by cursor movement; it would update very frequently, creating a lot of data, most of which doesn't need to be remembered. So, it is necessary to delete old values [25], but only those that definitely won't be needed any more since they can't be recovered. The obvious solution is to have reactives delete values from before that which is required by any present subscriber. However, this doesn't work in the case of a dynamic topology. Consider the case of a *Subscribe* message which experiences a considerable network delay. Another reactive may suddenly require those old values, without the parent even being aware of it.

The solution employed by Chronos is to use a *global* deletion threshold. Each sink knows the timestamp of the most recent value fed to it; this is the time threshold for *that* sink. *Every* ancestor of that sink will have values that are at least this recent, and possibly more so. Every relevant reactive is the ancestor of some sink. So, the global time threshold is the minimum sink threshold of all the sinks in the network, since *every* reactive in the network must be updated to at least this point. Thus, values older than this can safely be removed from each reactive's TDVs (U and V) and masks (C, P, and W).

What follows is a decentralised method for computing the global threshold T_G across multiple devices connected in a tree-structured network (which is acceptable for a client-server model). The minimum sink threshold for the sinks on just one device defines the *local* threshold T_L for that device. For some subtree of the network graph, the *tree* threshold is the minimum of each of the local thresholds for devices in that subtree. Each device tracks (a) its own local threshold and (b) the tree threshold for each of its neighbouring subtrees. Periodically, each device D sends to each of its neighbours N_i its own tree threshold *relative to* the edge E_i that joins them. This is the threshold of the subtree containing D that results from deleting E_i and is denoted T_i'. This is calculated by finding the minimum value of its own local threshold and each of its adjacent tree thresholds T_k, *except for* the tree threshold T_i corresponding to that neighbour:

$$T_i' = min(T_L, T_0..T_{i-1}, T_{i+1}..T_{n-1}) \tag{6}$$

The global threshold is then given by the tree threshold of the entire network:

$$T_G = min(T_L, T_0..T_{n-1}) \tag{7}$$

The frequency with which thresholds are updated is a trade-off between space and time efficiency. Since the removal frequency can be much lower than the actual signal frequencies, very little overhead is introduced by this method.

When a better threshold is discovered, a device will send a message to each local actor, informing it of the new threshold and asking it to trim its TDVs to remove old values.

When a *new* reactive is created by a topology switch, it is not allowed to access values of other reactives from before the timestamp associated with the event that triggered its creation. Otherwise, new reactives might want to go digging around in old values that no longer exist. Consider a *fold* operation. The current value of a signal is defined by the culmination of events leading up to that point. This is normally fine since the value is updated incrementally. But a *newly created* folding reactive wouldn't be able to access these old values. Thus, semantically, folds should *only* include events that were triggered after their creation. The only alternative is to keep *every* value throughout history in memory, which isn't feasible.

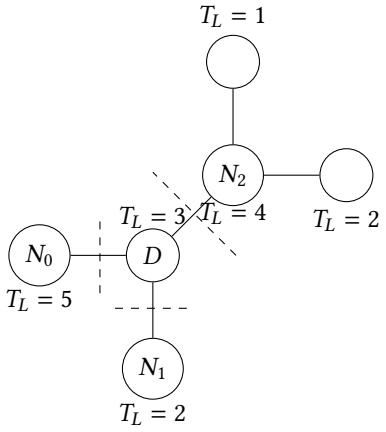


Figure 11. For device D, the global deletion threshold T_G is calculated by $min(T_L, T_0, T_1, T_2)$. $T_L = 3$, $T_0 = 5$, $T_1 = 2$, $T_2 = min(4, 1, 2) = 1$. Therefore, $T_G = min(3, 5, 2, 1) = 1$. This means it is safe for D to remove old values from $t \leq T_G = 1$.

4.3.12 Address Resolution.

If a reactive wants to subscribe to one of its parents on the same device, this is easy. The reactive will have a reference to the parent, which can be used to directly look up the actor for that reactive, or create the actor if it doesn't exist. However, for reactives on other devices, this isn't so simple. Reactives can't be serialised for inter-device transmission. This limitation stems directly from the fact that arbitrary functions can't be serialised in ScalaJS, or compared for equality. Thus, reactives themselves can't be serialised or checked for equality. Instead, an addressing system is introduced. Each sink is assigned a number. The parents of each reactive are also ordered. The address for a reactive is given by a path from some sink to that reactive. This is formed by a sink ID followed by a sequence of parent IDs. This system of addressing isn't unique, but it doesn't need to be. However, this will only work for static portions of the graph, consisting of reactives connected to at least one sink by a series of only static links. Reactives of this kind are called *statically-addressable*.

This idea can be extended to work with higher-order reactives by, for switching reactives, using not just the parent ID, but also the value from the static parent, which caused the dynamic link to be what it is. From this, a receiver is able to reconstruct the reactive being referred to. However, this can only work in cases where the switching value is serialisable. Thus, the requirement is added that any reactive which is the static parent of a higher-order reactive must have a serialisable value type.

Internally, each device uses reference-based equality checking to prevent duplication in cases where a single reactive has multiple addresses. Each device maintains two dictionaries: one from addresses to reactives (the address table) and another from reactives to actors (the actor table). An address lookup is performed whenever one actor needs to communicate with another and consists of the following steps:

1. Find the *longest prefix* of the address which is already in the address table. Starting at the reactive corresponding to this prefix address, traverse the dataflow graph according to the remaining portion of the address. For each reactive that is passed, add its address and value to the address table. If the original address is already in the table, this step amounts to doing nothing. After this step is completed, the original address will be in the table.
2. Given the reactive corresponding to the address, see if it has an actor in the actor table. If it doesn't, create a new actor for this reactive and add it.
3. Finally, return the actor reference.

4.3.13 Garbage Collection.

Old values aren't the only things that need to be removed. With a dynamic topology, entire reactives can become redundant. Since the dataflow graph is acyclic, reference counting is sufficient. If an actor has no children left, the reactive system is free to delete it. Upon deletion, it unsubscribes from all of its parents. However, this presents a rather serious problem: what if they are needed again at a later date? The actor can be reconstructed, but its state will be reset. This also isn't a problem, 'reset upon removal from the graph' is reasonable behaviour. The real problem is that it won't *always* be reset, and thus behaves inconsistently. For the state to always reset, the reactive would *depend on* its *children's* subscription masks, creating a circular dependency. This particularly affects the *fold* operator, since its *incremental* state is sensitive to the distinction between resetting and not resetting. At this point, Chronos offers no solution to the problem and simply accepts the quirk. In the few cases where this matters, the solution is to make sure deletion-sensitive reactives are all *statically-addressable* so that they are permanent. The only simple alternative is to keep old reactives around indefinitely, which is usually unacceptable.

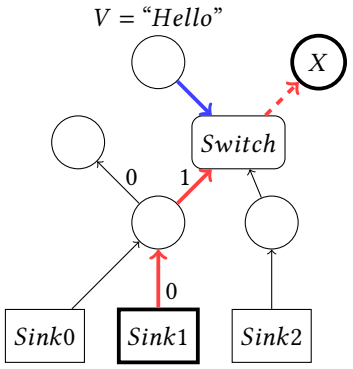


Figure 12. One possible address for the reactive X is $1 \rightarrow 0 \rightarrow 1 \rightarrow$ *"Hello"*, as indicated by the highlighted path. The first value is the sink ID. The subsequent values are the parent IDs. The switching reactive has to use the value from its static parent as the next parent ID.

5 Discussion

That completes the main contribution of this document; an exposition of the design decisions, problems, and solutions encountered in the development of Chronos, as they refer to the objectives previously stated. Chronos solves the dynamic-graphed multi-tiered FRP *engineering* problem. In doing so, it has been demonstrated that it is possible to maintain complete glitch freedom in the multi-tiered case of FRP, even when the dataflow graph is not static. Chronos solves glitch freedom by introducing TDVs with value *histories* to ensure mutual consistency of parent values. Then, old values which are no longer necessary are removed using a global deletion threshold. All of this is done without requiring any kind of global locking mechanism, allowing a multitude of propagations to take place *in parallel*, making maximal use of the available CPU bandwidth. Each device also does its own

local propagation without intervention. The server doesn't micromanage and only acts as an arbiter for access to shared state. Thus, Chronos meets each of the stated objectives. This *combination* of features and guarantees in a web environment represents a real advancement in the MT FRP paradigm. Multi-tier functional reactive programming, in this combined form, has seen very little use in practice. It is hoped that this project can serve as a small step towards changing that, given the enormous potential.

6 Conclusion

The concept of multi-tier functional reactive programming was explored. Reactive programming abstracts over the flow of time in an application. Multi-tier programming makes web development more integrated. An MT FRP framework called Chronos was designed for the web in the Scala programming language. Much of the latest in FRP techniques and design principles were used along the way. Chronos features higher-order reactives, simultaneous event propagation, and complete glitch freedom. A novel variation of QPROP was created in the process. These features make Chronos competitive (by feature parity) with the state-of-the-art in MT FRP, including other projects like Gavial and Eliom.

6.1 Future Work

The main concern with the current design is the inconsistent deletion semantics. An important research topic is to find a solution to this problem, whether that be by making the implementation more capable, or by limiting the range of programs that may be expressed to something that is provably consistent.

A notable omission from this report was any ability to build GUIs or interact with databases. These topics exist outside of the theoretical underpinnings of MT FRP, but nonetheless are an essential part of web development. Furthermore, the reactive paradigm presents an excellent opportunity to build GUIs *compositionally*. It would be an opportunity sorely missed if one didn't subsequently go on to construct a GUI library *on top of* Chronos for precisely this reason. From a practical standpoint, the building of some more infrastructure like this *around* Chronos is an important next step.

A Source Code

The source code for Chronos will be made available at: *https://github.com/SgtSwagrid/chronos*.

References

[1] Bob Reynders, Frank Piessens, and Dominique Devriese. *Gavial: Programming the web with multi-tier FRP*. KU Leueven. Feb. 2020. DOI: 10.22152/programming-journal.org/2020/4/6.

[2] Kennedy Kambona, Elisa Gonzalez Boix, and Wolfgang De Meuter. *An evaluation of reactive programming and promises for structuring collaborative web applications*. Vrije Universiteit Brussel. July 2013. DOI: 10.1145/2489798.2489802.

[3] Guido Salvaneschi et al. *An empirical study on program comprehension with reactive programming*. TU Dormstadt. Nov. 2014. DOI: 10.1145/2635868.2635895.

[4] Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. *A Survey of Multitier Programming*. Technische Universität Darmstadt. Sept. 2020. DOI: 10.1145/3397495.

[5] Bob Reynders et al. *Scalagna 0.1: towards multi-tier programming with Scala and Scala.js*. KU Leueven. Apr. 2018. DOI: 10.1145/3191697.3191731.

[6] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. *Distributed system development with ScalaLoci*. TU Darmstadt. Oct. 2018. DOI: 10.1145/3276499.

[7] Engineer Bainomugisha et al. *A survey on reactive programming*. Vrije Universiteit Brussel. Aug. 2013. DOI: 10.1145/2501654.2501666.

[8] Jonathan Edwards. *Coherent reaction*. MIT. Oct. 2009. DOI: 10.1145/1639950.1640058.

[9] Henrik Nilsson Guerric Chupin. *Functional Reactive Programming, restated*. University of Nottingham. Oct. 2019. DOI: 10.1145/3354166.3354172.

[10] Bob Reynders, Dominique Devriese, and Frank Piessens. *Multi-Tier Functional Reactive Programming for the Web*. KU Leueven. Oct. 2014. DOI: 10.1145/2661136.2661140.

[11] Atze van der Ploeg. *Monadic functional reactive programming*. Centrum Wiskunde & Informatica. Sept. 2013. DOI: 10.1145/2503778.2503783.

[12] Daniel Winograd-Cort and Paul Hudak. *Settable and non-interfering signal functions for FRP: how a first-order switch is more than enough*. Yale University. Aug. 2014. DOI: 10.1145/2628136.2628140.

[13] Conal M. Elliott. *Push-pull functional reactive programming*. LambdaPix. Sept. 2009. DOI: 10.1145/1596638.1596643.

[14] Joscha Drechsler et al. *Distributed REScala: an update algorithm for distributed reactive programming*. Technische Universität Darmstadt. Oct. 2014. DOI: 10.1145/2714064.2660240.

[15] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. *Distributed Reactive Programming for Reactive Distributed Systems*. Vrije Unversiteit Brussel. Feb. 2019. DOI: 10.22152/programming-journal.org/2019/3/5.

[16] Gabriel Radanne, Vasilis Papavasileiou, and Jérôme Vouillon. *Eliom: tierless Web programming from the ground up*. IRIF, Paris Diderot University. Aug. 2016. DOI: 10.1145/3064899.3064901.

[17] Evan Czaplicki and Stephen Chong. *Asynchronous functional reactive programming for GUIs*. Harvard University. June 2013. DOI: 10.1145/2491956.2462161.

[18] Jeff Horemans, Bob Reynders, and Dominique Devriese. *Elmsvuur: A Multi-tier Version of Elm and its Time-Traveling Debugger*. 2018. DOI: 10.1007/978-3-319-89719-6_5.

[19] Florian Myter et al. *I now pronounce you reactive and consistent: handling distributed and replicated state in reactive programming*. Vrije Universiteit Brussel, Belgium. Nov. 2016. DOI: 10.1145/3001929.3001930.

[20] Alessandro Maragra and Guido Salvaneschi. *We have a DREAM: distributed reactive programming with consistency guarantees*. Vrije Universiteit Amsterdam. May 2014. DOI: 10.1145/2611286.2611290.

[21] Guido Salvaneschi, Joscha Drechsler, and Mira Mezini. *Towards Distributed Reactive Programming*. Technische Universität Darmstadt. 2013. DOI: 10.1007/978-3-642-38493-6_16.

[22] *Cats: Lightweight, modular, and extensible library for functional programming*. Typelevel. URL: https://typelevel.org/cats/.

[23] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. *43 years of actors: a taxonomy of actor models and their key properties*. Vrije Universiteit Brussel. Oct. 2016. DOI: doi.org/10.1145/3001886.3001890.

[24] *Introduction to Actors*. Lightbend. URL: https://doc.akka.io/docs/akka/current/typed/actors.html.

[25] Atze van der Ploeg and Koen Claessen. *Practical principled FRP: forget the past, change the future, FRPNow!* Chalmers University of Technology. Aug. 2015. DOI: 10.1145/2784731.2784752.