Croquetify: Transforming Single-User Functional Reactive Programs into Multi-user Applications

Yoshiki Ohshima ✉
Shizuoka University
Los Angeles, USA
Independent Software Engineer
Los Angeles, USA
Yoshiki.Ohshima@acm.org

Aran Lunzer
Independent Software Engineer
Los Angeles, USA
aran@acm.org

David A. Smith
axona.net
Cary, USA
david@axona.net

Abstract

Building robust real-time collaborative applications — "multi-user apps" — is hard. State transitions that are trivial in the single-user case become surprisingly complex once concurrent participants are involved, often requiring custom interaction protocols and tailor-made speculative execution and reconciliation mechanisms. Errors are harder to reproduce, and the application must still respond within milliseconds. AI coding agents do not yet handle these demands reliably.

In this paper, we present a novel way to develop multi-user apps. We employ a combination of a Functional Reactive Programming language called Renkon, which is a general-purpose language for application development with rich meta-programming features, and a network application framework called Croquet. We devised a program transformation that allows the same Renkon program to run as a single-user app or to use Croquet to run as a multi-user app without writing application-specific server-side code. This is done without losing Renkon's live programming capability.

To demonstrate how well this approach works for a practical application, we applied it to a real-time interactive programming environment called Renkon-pad. We show that not only can we transform a single-user programming environment into a multi-user one, but also that the multi-user environment can be used to develop a multi-user app in a live programming manner. This experiment supports the idea that the transformation is applicable to general and complex programs, with the programmers' intent in the original program preserved.

CCS Concepts: • **Networks** → **Programming interfaces**; • **Software and its engineering** → **Integrated and visual development environments**; **Data flow languages**.

Keywords: Croquet, Renkon, Renkon-pad, functional reactive programming, programming environment, multiuser application, program transformation

ACM Reference Format:
Yoshiki Ohshima, Aran Lunzer, and David A. Smith. 2026. Croquetify: Transforming Single-User Functional Reactive Programs into Multi-user Applications. In *Proceedings of the 2026 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '26), October 4–9, 2026, Oakland, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3840586.3843209

This work is licensed under a Creative Commons Attribution 4.0 International License.

Onward! '26, Oakland, CA, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2934-8/2026/10
https://doi.org/10.1145/3840586.3843209

1 Introduction

Building robust real-time collaborative applications ("multi-user apps"), such as collaborative design tools or real-time action games, is hard. State transitions that are trivial in the single-user case become surprisingly complex once concurrent participants are involved, often requiring custom interaction protocols. For instance, one can imagine a simple case where more than one `pointerEnter` event occurs before a `pointerLeave` event. Such a case may require a tailor-made speculative execution and reconciliation mechanism. Reproducing an error is harder. On top of these issues, the app needs to be optimized to provide responsive interaction, down to a few milliseconds of latency.

AI coding agents are not a reliable solution yet. There are not enough examples to train on for them to avoid common and uncommon pitfalls. AI agents have more difficulty if the development and deployment of an app require an application-specific server, as setting up and updating the server is cumbersome and error-prone. We also advocate that a real-time collaborative application should be created and tested collaboratively, but the methodology has not been well developed.

In this paper, we present a new way to develop multi-user apps. We employ a combination of a Functional Reactive Programming (FRP [9]) language called Renkon [19] and a network application framework called Croquet [11], and show that apps created with this combination are robust, performant, and cleanly implemented.

Notably, we devised an automatic program transformation that allows us to run the same Renkon program as a single-user or a multi-user app. This is done without losing the nice properties of Renkon, such as its live programming capability. While the resulting multi-user app may not have all desirable multi-user features initially, developers can gradually add more features while maintaining the clarity of the code.

To demonstrate how well this approach works for a practical application, we applied it to a real-time interactive programming environment called Renkon-pad. Renkon-pad lets a group of users create single- and multi-user apps together. Moreover, the fact that Renkon-pad is written in itself and can be dynamically improved further stresses the point that the combination is a framework for creating practical applications.

In Figure 1, the screenshots show two browser tabs running multi-user Renkon-pad, editing Renkon-pad's own code. The smaller "runner" window in them runs the Renkon-pad being edited. The nested Renkon-pad itself is also multi-user in a separate session. The translucent greenish overlay visible in the screenshot indicates the other participant's viewport.

Renkon is a general-purpose FRP language for building various kinds of applications. Its core semantics follow those of typical FRP languages, which have been shown to provide a basis for the clean organization of application code. The evaluation of a Renkon program is built upon the concept of logical time, which is typically synchronized with the real-time clock. This provides a foundation for reasoning about a real-time program, especially when network and user events are involved. While Renkon draws upon past FRP languages, it differs in that a Renkon program is a set of loosely coupled reactive node definitions. This facilitates modifying the program at runtime and makes program transformation and generation as simple as concatenating node definitions.

Croquet is a network architecture based on the idea of replicated computation. It also uses the concept of explicit logical time. The architecture employs the "reflector" (a network server on the public internet), which coordinates the participants but does not compute the application state. Its main function is to route the events coming from participants. The reflector advances the application's logical time, and the application state on all participants' computers transitions *bit-identically*. This allows a multi-user app to be developed only by writing client-side code. The latency in interaction in a Croquet-based app tends to be very low. In a favorable setting, it can be within a few milliseconds, determined only by the distance to the reflector.

The contribution of this paper is to demonstrate that the combination of Renkon and Croquet yields a clean framework for writing collaborative applications. Specifically, we will show that:

- The same program can be executed as both a multi-user and a single-user app via a program transformation.
- Live programming a multi-user app is possible and useful.
- Managing the application state in FRP and avoiding a server that computes the application state simplifies multi-user app development.
- A collaborative programming environment can be written in itself.

In addition to the design and implementation of the particular system we describe, we attempt to distill the key factors that make this approach attractive.

The rest of the paper is organized as follows. In Section 2, we give an overview of the Renkon language. In Section 3, we explain the Croquet network architecture. In Section 4, we describe a simple program transformation called "croquetify" to transform a Renkon program to be multi-user over Croquet. In Section 5, we describe multi-user Renkon-pad as a larger practical example written in this framework. In Section 6, we mention related work. In Section 7, we discuss our findings. Section 8 concludes the paper.

Note to readers:

The Renkon-pad application is open source. To test the version submitted for review, open a URL such as https://tinlizzie.org/onward2026/?q=12345#pw=1, but replace "12345" in the `q` parameter with a random string of your choice. Make sure that your tabs are visible so that the browser doesn't put them to sleep. The latest version of Renkon-pad is available at https://yoshikiohshima.github.io/renkon-pad. You can load the `index.renkon` file into Renkon-pad to see the implementation of Renkon-pad itself.

By default, your session will use a Croquet reflector in Northern California, and the distance to it dictates the latency you experience. If you are on the West Coast of the US, the latency is almost unnoticeable, but it will feel laggy otherwise. You can set up your own reflector locally because Croquet is open source; in that case, the latency becomes virtually zero.

2 Renkon

Renkon is a programming language that follows the original formulation of Functional Reactive Programming (FRP) by Elliott et al. [9]. A program in Renkon consists of a set of reactive nodes. In general, each node reacts to input changes and produces an output. In turn, other nodes that depend on the output react to it. In other words, the program nodes form an acyclic graph connected via dependency relationships. Inputs to the graph are propagated through the graph to compute the application state.

The Renkon language runtime consists of the transpiler, the analyzer, and the evaluator. The implementation is written in TypeScript so that it can be used in the browser as well as in Node.js.

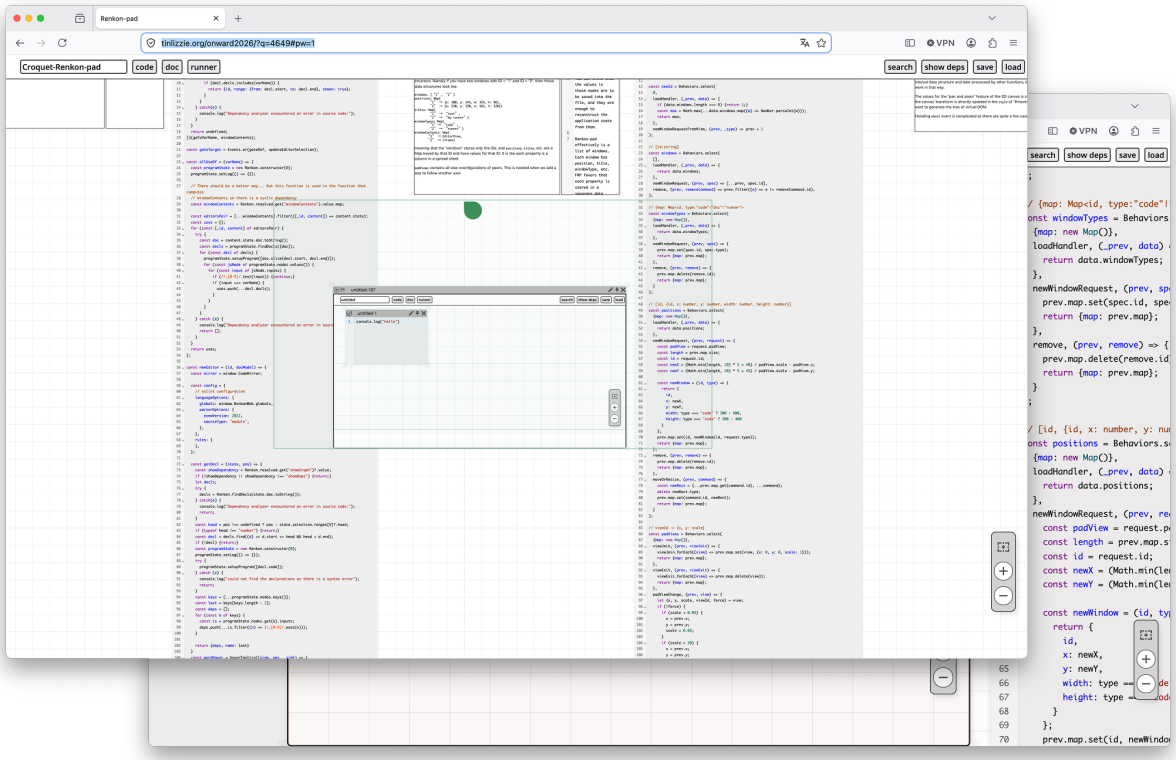

Figure 1. Two browser windows in the same session, editing Renkon-pad's own code and testing itself in the embedded runner. The green translucent overlay shows the region seen by the user of the second window; the green pointer, currently near the overlay's top edge, represents that user's cursor.

Renkon's surface syntax is a large subset of JavaScript and TypeScript; most JavaScript syntax is valid Renkon syntax. All JavaScript values are valid values in Renkon, except `undefined`, which Renkon uses to signify the absence of a value. The Renkon transpiler (built using the Acorn parser [16]) takes a textual node definition and creates a JavaScript expression that evaluates to an internal data structure of a Renkon node.

The Renkon transpiler works on each node in isolation. It analyzes the definition and notes all external references (free variables) as the names of the node's dependencies. Because the transpiler works on one node at a time, the nodes are loosely coupled, and the programmer can change the definition of a node independently of other nodes, even at runtime. Also, the order of the definitions in a program text makes no difference to the meaning of a program; it means that a program can be easily represented in a non-textual form (a boxes-and-wires diagram, for instance), and this property is conducive to program transformations. A node can be bound to a top-level variable, and the variable name becomes the name of the node.

The Renkon analyzer topologically sorts the nodes based on the dependency relationship and creates a linear list of nodes for evaluation. The transpiler and analyzer need to run only when a node definition changes.

The Renkon evaluator runs when a user event or some other external event enters the dependency graph. The evaluator walks through the list, invokes the expression of each definition with updated values as necessary, and stores the computed value in the runtime structure. This evaluation scheme ensures that a node is evaluated at most once at each logical time t, and the program does not exhibit temporary inconsistent evaluation, or *glitches*, when a topological sort exists. That is, when all nodes in a dependency graph can be sorted topologically, it is guaranteed that the dependencies of a node have their values for the logical evaluation time t; therefore, the value of the node will be consistent with the values of its dependencies.

Renkon has a few notable features that help write application programs easily and interactively.

- Drawing upon the original FRP formulation, time-varying discrete events and continuous values are separated.
- Promises and generators are integrated into the language so that a resolving promise is treated as a change on a node.

- The definitions of a reactive node can be changed at runtime.

Separating values on discrete and continuous time domains provides a clean way to express the intention of how a value will be used. This also helps the program transformation we will describe in Section 4.

Renkon is good for writing graphical applications. While Renkon is completely agnostic to the choice of graphics library (including "vanilla" JS with direct DOM manipulation), it is convenient to use a virtual DOM library. A virtual DOM library allows us to treat DOM elements as functional values, and the program stays functionally pure until the side-effectful screen updates that happen at the end of the evaluation cycle. We find that Preact [6] is a good match for Renkon.

By treating JavaScript promises as time-varying values, the language accommodates a wide range of JavaScript built-in features and third-party libraries. For instance, the `import` expression loads third-party modules, which can be seamlessly integrated into a practical program.

Let us look at a simple program that uses DOM elements.

```
1 const {h, render} = import(
2   "./preact.standalone.module.js");
3 const collection = ["a", "b", Events.timer(20)];
4 const dom = h("div", {}, collection.map(
5   (word) => h("span", {}, word)));
6 render(dom, document.body);
```

Note to readers:

You can paste the above code into the Renkon-pad code editor (remove line numbers if they are also copied), instantiate a runner, and press the play button in the runner. Note that the program being built in Renkon-pad runs in single-user mode by default; instances of the above program appearing in different runner windows are not connected to each other.

In the program above, we import `h` and `render` from the unmodified Preact library. The JavaScript built-in `import` expression returns a promise, and the Renkon evaluator updates `h` and `render` when the library is loaded (line 1). The `h` function of Preact creates the DOM tree definition as an object tree, and the `render` function of Preact "reconciles", or calculates the difference, between the provided virtual DOM object tree and the actual DOM target.

The `collection` node (line 3) has a value that is an array with three values `"a"`, `"b"`, and the result from `Events.timer(20)`. `Events.timer()` yields the elapsed time since the start of the session. In this example, the event stream contains numbers that are multiples of 20. Because the timer fires every 20 milliseconds, the value of `collection`, which depends on the timer's value, is updated at the same rate.

The `dom` node (lines 4–5) is a `div` node that has three `span` elements, each of which is an element of `collection`.

The node with the `render` call (line 6) is evaluated whenever `dom` is updated. (Strictly speaking, it is also evaluated when `render` is updated, but that happens only when the program is modified. The variable `document` is a free variable on line 6, but it is treated as a known global constant.) Notice that line 6 does not have a name and is not assigned to a top-level variable. This is one way to reveal the programmer's intention that no other node will depend on this expression.

Live programming capability comes naturally in Renkon. The transpiler treats the definition of, for example, `dom` in isolation and finds that `h` and `collection` are free variables. However, it does not look at what is bound to them. The definition of `dom` can be changed between evaluation steps. When this occurs, the transpiler and analyzer run, and after that, the program continues.

2.1 Events and Behaviors

A Renkon node represents a stream of time-varying values, and it is either an "Event" or a "Behavior", a distinction that is drawn from the original formulation of FRP. An Event has a value only at the logical time when an external action (user interaction, network activity, etc.) occurs. It does not have a value (indicated by its value being `undefined`) at other times. A Behavior has a value "all the time". It is commonly used to represent a value in persistent application state.

A primitive Event or Behavior is created with a combinator method on the `Events` class (such as `Events.timer`) or the `Behaviors` class, respectively. Also, a constant value such as `10` is a Behavior. When the Renkon analyzer analyzes a node, its type is determined by the types of its dependencies. In general, when any one of the dependencies is an Event, the node is an Event. Consider the following definition:

```
1 const e = {t: Events.timer(1000), x: 10};
```

In this case, the reactive node bound to `e` is an Event, even though "10" is a Behavior whose value is constantly 10. The node `e` gets a value (the value is an object with properties `t` and `x`) only at the times when the `Events.timer` node has a value.

2.2 Combinators

There are "FRP combinators" that provide more sophisticated dataflow beyond the simple dependent update rule. See documents on Renkon, such as [19] and [18], for more details. We touch upon a few relevant ones below.

The `select` combinator uses its "previous value" (see Section 2.7) and the incoming event's value to compute its new value. It is analogous to the `fby` expression in languages such as Lucid [21], but it allows multiple cases. It can also be seen as an "actor"-like construct; namely, a message it receives selects the handler to be invoked, and the handler computes the new value.

Typically `select` is used in this form:

```
1  const collection = Behaviors.select(
2    [],
3    reset, (_prev, _reset) => [],
4    timer, (prev, timer) => [...prev, timer]);
```

The `collection` node (line 1) begins with the initial value, in this case an empty array `[]` (line 2). Whenever the node bound to `timer` fires, `collection` becomes a new array with the new event added (line 4). Therefore, `collection` is updated to `[0]`, `[0, 1000]`, `[0, 1000, 2000]`, and so on every 1,000 milliseconds (when `timer` is `Events.timer(1000)`). When the `reset` event occurs, `collection` is restored to an empty array (line 3).

2.3 Send and Receiver Combinators

In writing an interactive application, a situation arises when a UI element is created based on the application state, and the UI element needs to affect the same application state when a user interacts with the UI. This forms a cycle in the dependency graph; thus, it has to be handled properly to make the FRP-based system practical while keeping clean semantics.

Let us take the following program as an example:

```
1  const {h, render} = import(
2    "./preact.standalone.module.js");
3  const reset = Events.receiver();
4  const timer = Events.timer(1000);
5  const collection = Behaviors.select(
6    [],
7    reset, (_now, _reset) => [],
8    timer, (prev, timer) => [...prev, timer]);
9
10 const resetter = ()=>Events.send(reset, "doIt");
11 const buttonDOM = collection.map((n) =>
12    h("button", {onClick: resetter}, n));
13 const buttons = h("div", {}, buttonDOM);
14 render(buttons, document.body);
```

In the program above, `collection` holds past timer values in an array (lines 5–8), as in the previous code example. The `buttonDOM` array (lines 11–12) consists of `button` elements, each labeled with a value from `collection`. This means that `buttonDOM` depends on `collection`. The button also has an `onClick` handler that resets the array, meaning that `collection`'s state depends on `buttonDOM` via the "reset" event. If we try to make `collection` a direct dependency of `buttonDOM`, a cycle would form, and the program would not have a sound evaluation order.

Our solution, already used in the example, is the "send" and "receiver" mechanism, drawn from Flapjax [17]. The `Events.send` call (line 10) sends a value to a `receiver` (in this case, `reset`, defined on line 3). The key point is that the sent value is delivered in the next evaluation cycle, at a different logical time. This ensures that a node has at most one value at a time, breaking the cyclic dependency.

A `receiver` can be either an Event or a Behavior. The common case is the Event type, but a Behavior variant is useful when the value is expected to persist — for instance, initialization logic that acquires a value over the network and uses it throughout the application afterward.

2.4 The Use of Immediately Invoked Function Expression

Sometimes a reactive node needs to compute a value that requires more than a simple expression — perhaps using a temporary variable, or a control structure such as `if` or `for`. Renkon allows the use of an Immediately Invoked Function Expression (IIFE) to compute a value. For example, one can write a reactive node `col3` that depends on `col1` and `col2` in this way (a convoluted example to show the syntax; what it does is irrelevant):

```
1  const col3 = ((c1, c2) => {
2    const result = [];
3    for (let i = 0; i < c1.length; i++)
4      if (c1[i] === c2[i]) {result.push(i);}
5    return result;
6  })(col1, col2);
```

The expression is invoked when the dependencies (`col1` or `col2` on line 6) change. On invocation, the formal arguments (`c1` and `c2`) are bound to the values of the nodes, just as in JavaScript. The value returned becomes the value of the entire reactive node `col3`. The value `result` is mutated during execution, but as long as it does not leak and is not used elsewhere, functional purity is maintained.

This improves the expressiveness of Renkon, because sometimes it is simply easier to use a control structure to compute a value, and some third-party libraries require imperative setup. The use of an IIFE admittedly looks "iffy", as it mixes imperative language constructs into a functional setting, but it is very useful for writing a practical application with complex data structures. As long as the internal mutable data is contained within the body of the function, it does not violate the pure nature of FRP.

2.5 Handling Mutable State

Most third-party libraries use mutable objects. Also, basic data structures such as `Array` and `Map` are mutable. It is convenient to use them as they are, rather than working around them in awkward ways, such as needing to make a new `Map` for each mutation in React [22].

In Renkon, a common idiom is to keep the same mutable object but wrap it in another plain object:

```
1  const pos = Behaviors.select(
2    {map: new Map()},
3    move, (prev, move) => {
4      // move:<{id:string, x:number, y:number}>
```

```
5       prev.map.set(move.id, move);
6       return {map: prev.map};
7     });
```

Let us say that pos represents the positions of graphical shapes, keyed by the shapes' ids. It keeps the values in a Map, and when the move event happens, the Map is mutated by the set method call on line 5. But the actual result returned from the function is a fresh object with the same, but mutated, Map in the map property.

This works nicely. The mutable state is internal to the node. Its dependencies are notified because the value for pos is a fresh object. As long as the internal mutable data structure (such as this Map) is not mutated outside of the node, the functional purity is maintained. When a node that generates virtual DOMs reacts to this change, developers have control over how to check the values in the Map. We use this technique throughout applications such as Renkon-pad.

2.6 DOM Event Handling

Because DOM user events are mutable and some methods on an event have to be called in the immediate event handler, the canonical form of handling DOM events is:

```
1 const down = Events.listener(
2   anElement,
3   "pointerdown",
4   (evt) => {...});
```

This sets up a function (line 4) as the direct event handler on the actual DOM element. The body of the function can call methods such as preventDefault() and compute a value from the event. The value returned from the function becomes the new value of the Event bound to down.

Note that Events.listener is reactive to value changes. When the third argument is a free variable that points to a node whose value is a function, the handler function bound to the actual DOM is automatically updated. In this way, a use case that requires a higher-order stream in other languages can be described cleanly.

2.7 One More Note on Time

The semantics of FRP are clean, but implementing it on top of a JavaScript engine requires some consideration. In theory, there would be no "previous" value for a Behavior if it were on a truly continuous time domain. Also, two causally independent Events never occur at the same instant.

In practice, the Renkon implementation uses JavaScript's discrete evaluation mechanism, and Behaviors act as step functions. This also means that more than one event can occur within the same inter-cycle gap, and these events are "delivered" at the next Renkon evaluation cycle. For the program to handle multiple values, a receiver may be defined with the {queued: true} option. In this case, multiple events are queued for the next evaluation cycle and delivered in an array.

2.8 Meta-Programming

Renkon provides deep introspective features. The entire state of a running program, including the original node definitions, transpiled results, the evaluation order of nodes, and current node values, is stored in an instance of a class called ProgramState, and is visible from the program itself as the Renkon global variable. The updateProgram method on a ProgramState takes new node definitions and updates the internal dependency graph.

ProgramState has a method called registerEvent() to "inject" a new value into a receiver node from outside. Thus, for a JavaScript program hosting a ProgramState instance, the following JavaScript code first sets a value (true) into the node named reset (line 1) and runs an evaluation cycle by calling evaluate (line 2).

```
1 programState.registerEvent("reset", true);
2 programState.evaluate(Date.now());
```

3 Croquet

Croquet is an open-source network application framework for creating real-time multi-user apps [11][27][1]. It differs from typical collaborative frameworks that employ a client-server architecture, where the server computes the application's state and sends it to participants. In contrast, the application state of a Croquet app is computed in the JavaScript interpreter on the client side. It needs only a small server called a "reflector", which receives each event sent by any participant and reflects it back to all participants.

Let us look at Figure 2 and follow a typical flow of execution. We will explain what the "model" and "view" are in more detail below; for now, just consider that the model is the shared state of a multi-user app, and the view consists of the browser or other runtime features that reside outside of the model. When a user interacts with the app (say, clicking on a DOM element on the screen), the resulting event is sent to the reflector (step 1 in the figure). The reflector puts a timestamp on it and sends it back to all participating clients (2). Upon receiving the event, the model runs, and the model on all participating clients transitions to the new state, "bit-identically" (3). Each client's view then updates its own screen and other browser-dependent externals (4). The view has direct read-only access to the data in the model at this step, as the model and the view are actually in the

[1]While the company that originally developed the Croquet framework ceased operations, the authors believe that the underlying ideas remain powerful and the open-source implementation is readily usable. Any organization or individual can set up their own reflectors locally or globally. There were also companies that created slightly modified versions of Croquet under different names, but for this paper these extensions are not relevant, and we use the name "Croquet" throughout the paper.

same JavaScript execution context (boxes labeled client A and client B).

The communication between a client and the reflector uses a WebSocket or an order-preserving WebRTC data channel so that the order of messages is preserved once the reflector receives them and puts timestamps on them.

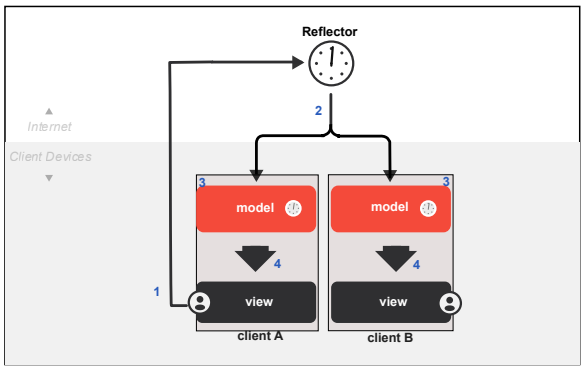

Figure 2. A diagram of the Croquet architecture. 1. The user on client A interacts with the app and an event is sent to the reflector. 2. The reflector puts a timestamp on the event and sends the event back to all participants. 3. The handler for the event is executed in a bit-identical manner. 4. The view updates the screen by reading the model data without going through the network.

The advantage of moving computation from the server to the client is that the reflector can be fully application agnostic. This means that the same reflector can serve different sessions of different applications simultaneously, without adding expensive compute resources to the server. Debugging an application is easier, as the shared data, which might be on the server in a client-server setting, is visible directly in the developer console on the client. Also, the reflector operates without interpreting any data that clients create, thus allowing end-to-end encrypted shared applications. The capacity of a reflector, such as the maximum number of users supportable in a shared session, depends on the complexity of the application. We found that dozens of highly active users can participate in a real-time 3D virtual world environment [20] (as long as the clients' rendering can keep up), while the same session can also accommodate hundreds of spectators. There can be even more users in a session of a less intensive application.

How does Croquet guarantee bit-identical execution in the replicas of the model? First, Croquet uses JavaScript as its foundation. The JavaScript language standard ensures a high degree of conformity in execution semantics across different platforms. Where the standard allows differences, we provide our own implementations that ensure the same outcome on every platform (in particular, for transcendental functions such as trigonometric functions). We install a seeded random number generator in place of `Math.random`; this allows client code to call `Math.random` transparently to obtain replicated random numbers. Croquet also replaces methods of `Date` to avoid obtaining different values.

Second, we employ a strong notion of model-view separation. The Croquet model contains all information about the logical state of the application, and must be amenable to serialization and deserialization. The view is everything else, which may include DOM elements, the `document` object, WebGL and Audio contexts, etc. The view may be fully reconstructed from the current state of the model when necessary. Note that the terms "model" and "view" do not imply non-graphical and graphical state. Any data structure that is to be shared among participants is in the model, and every other part of the application is in the view. For instance, a multi-user shape editor would have the location, stroke style, fill color, etc. of a shape in the model so that they are shared among users, even though they concern visual representation. Also note that accessing view-specific values such as `window.innerWidth` from the model will break bit-identical execution. Developers need to write the model code to use logical coordinates, for instance, and the view code to map the logical coordinates to the actual screen coordinates.

The communication between the model and the view uses the Croquet library methods `publish` and `subscribe`, which send and receive Croquet events. As shown in Figure 2, the application view code calls the `aView.publish()` method to send a user event to the reflector, and the reflector sends it back to all participants to trigger the model-side handler registered with `aModel.subscribe()`. This is called a view-to-model event. On the other hand, when the model notifies the view to update the screen (or play a sound, etc.), the event does not go over the network. When the model executes `aModel.publish()`, each replica executes the same call during deterministic model execution, triggering the handler registered with `aView.subscribe()` locally.

A practical multi-user app may run for a long time, even months or years, with users joining and leaving many times (imagine a shared document editor for writing a conference paper). To support such an application, Croquet takes a "snapshot" of the model state from time to time (by default, every 5 seconds of model execution time). When a user joins an existing session, the latest snapshot plus outstanding events since the snapshot are delivered to the "latecomer" browser tab so that the new client can catch up to the identical state with other users.

The snapshot is saved to a file server associated with the reflector. Having a file server allows Croquet to store snapshots as well as large assets for an application. For example, a 3D virtual world application may store a large 3D asset on the file server, and the Croquet model only stores the URL to the asset. The Croquet view on each client loads the 3D

```
1  class MyModel extends Croquet.Model {
2    init() {
3      this.count = 0;
4      this.subscribe("counter", "reset",
5                     this.resetCounter);
6      this.future(1000).tick();
7    }
8    resetCounter() {
9      this.count = 0;
10     this.publish("counter", "changed");
11   }
12   tick() {
13     this.count++;
14     this.publish("counter", "changed");
15     this.future(1000).tick();
16   }
17 }
```

```
18 class MyView extends Croquet.View {
19   constructor(model) {
20     super(model);
21     this.model = model;
22     countDisplay.onclick = event=>this.counterReset();
23     this.subscribe("counter", "changed",
24                    this.counterChanged);
25     this.counterChanged();
26   }
27   counterReset() {
28     this.publish("counter", "reset");
29   }
30   counterChanged() {
31     countDisplay.textContent = this.model.count;
32   }
33 }
```

Figure 3. The model and view code of a self-incrementing counter.

asset from the server, but the model is concerned not with the large data, but with the orientation, scale, etc.

The Croquet data serializer handles cyclic data structures and supports standard objects such as `Map` and `Set`. Thus, the model data structure does not have to be written in awkward styles just to satisfy the requirements of the framework. In addition to support for JavaScript's built-in classes, developers can provide pluggable reader and writer functions for third-party classes. This allows the model to contain, for example, the `Text` object of the CodeMirror text editor or Three.js numeric classes such as `Matrix4`. The developer can also write out "persistent data" to support program code evolution while keeping the essential part of the application state.

Recall that the reflector sequences events coming from different users so that the order of events that a client sees is guaranteed to be the same for all clients. In essence, the replicated model treats each event as a trigger for a transaction, and developers can ensure the integrity of the application semantics (see Section 6 for comparison to "merge-based" collaboration libraries).

The reflector also generates timer events, or "ticks", as needed, so that a real-time application with animation, for example, is synchronized across the participating computers even when no user is actively generating events. This means that Croquet does not require a distributed synchronized clock, because the reflector advances the logical clock and notifies the participants.

The fundamental idea of Croquet described above appears to be naive and simple (and its simplicity gives it power). However, real-world networks suffer from problems such as intermittent WiFi or users simply closing their laptops. The core developers of Croquet have paid careful attention to these details. As a result, the Croquet implementation is well tested in deployed practical applications and has proven to work reliably.

Figure 3 is a simple Croquet counter example written in vanilla JavaScript. This application shows a value on the users' screens (via `countDisplay.textContent = ...` on line 31). This app increments the count every 1,000 milliseconds and resets it to zero when any one of the users clicks on the display.

The `MyModel` class has a property called "count" that constitutes this app's entire logical application state (line 3). An event from the view to the model called `reset` in the `counter` scope is sent when a user clicks on the `countDisplay` DOM element (line 22), and is delivered to all replicas of the model via the reflector. In addition to the reset event, the model has a (replicated) future event loop that runs once every 1,000 milliseconds (line 15). The model publishes a model-to-view event called "changed" when the `count` value changes (lines 10 and 14), and the handler for the event in the view updates the `textContent` of the DOM.

Note again that the application does *not* need to propagate the value of `count` across the network. The tick events created by the reflector, and each user event requesting a counter reset, are sent over the network; each replica of the model independently computes the next value of `count`, in a bit-identical manner.

A Croquet application can be written in a different surface language. Indeed, in the next section, we show that a Renkon program can be "croquetified".

4 Croquetify: Generating a Multi-user Program

In this section, we describe the program transformation that converts a single-user Renkon FRP program into a multi-user program. We call the transformation "croquetify".

We use the counter program shown in Figure 4 to explain the transformation. The `counter` node (lines 1–3) in the program starts with the value zero (line 2). When the `change` event (line 4) fires, the new value becomes the sum of the

```
1  const counter = Behaviors.collect(
2    0,
3    change, (prev, change) => prev + change);
4  const change = Events.or(incr, decr, thousand);
5  const timer = Events.timer(1000);
6  const thousand = timer ? 1000 : undefined;
7
8  const incr = Events.listener(
9    document.querySelector("#incr"),
10   "click",
11   (evt) => 1);
12 const decr = Events.listener(
13   document.querySelector("#decr"),
14   "click",
15   (evt) => -1);
16 document.querySelector("#out")
17   .textContent = counter;
```

Figure 4. The counter program in Renkon. It has one integer value, which is incremented by 1,000 each second and also changes when the user presses buttons called incr and decr.

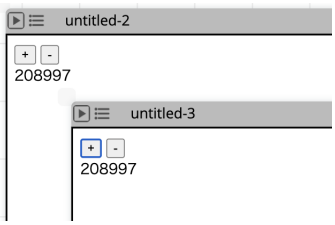

Figure 5. A screenshot of the counter program (shown in two runner windows). The "+" and "-" buttons are named incr and decr, respectively. Every 1,000 milliseconds, the counter value is incremented by 1,000, while pressing a button changes the value by 1 or -1.

previous value and the change event (line 3). The change event fires when one of `incr`, `decr`, *or* `thousand` fires.

Recall that the textual order of node definitions makes no semantic difference. The change node is textually placed after the definition of `counter`, for instance. The transpiler works on each node in isolation, and then the analyzer topologically sorts them regardless of their textual order.

The nodes `incr` and `decr` are DOM event handlers on elements named "incr" and "decr" (lines 8–15), and they take a value of 1 or -1 when the event occurs (lines 11 and 15, respectively). The node on line 16, which is a property assignment expression, is triggered when `counter` updates and effectively makes the content of the DOM element called "out" causally connected with `counter`.

When we run this program, we see something like one of the windows in Figure 5. (The screenshot already shows the multi-user case.) The `thousand` node fires every 1,000 milliseconds, so the counter goes up by 1,000 each second, but when the user clicks on the "+" or "-" button, the value is changed by 1 or -1, respectively.

Now, we show how we transform ("croquetify") the program into a multi-user one that uses Croquet. Recall that the Croquet mechanism employs the concept of model-view separation. The basic idea of croquetify is to split the reactive nodes into two groups, one for the model and another for the view.

In our counter case, `counter` is the fundamental application data. Also, `timer` is a replicated model event, and `change` and `thousand` should be in the replicated model. We can therefore think of a sub-program that consists of `counter`, `change`, `timer`, and `thousand` as being grouped into the model.

We also create another sub-program with all remaining nodes and put them into the view. The croquetify function's job is to create these two sub-programs from the original program. To specify the grouping, we explicitly pass the list of names, called the "modelRealm" spec, to the croquetify function.

After splitting the code, we need a way to trigger the model sub-program to run when a user event or timer event is delivered to the model from the reflector. Also, we want the view sub-program to run when the model changes so that the screen is updated. To achieve this, we add a few more nodes to each sub-program.

The croquetify function analyzes the original program and generates additional nodes to accommodate dependency relationships that cross the model-view boundary. For a view-side node that is used in the model (such as `incr` in the example), croquetify adds an `Events.receiver()` node to the model sub-program. In our case, the following two nodes are added:

```
1  const incr = Events.receiver();
2  const decr = Events.receiver();
```

For a model node that is used in the view (`counter` is the only one in this example), croquetify adds a `Behaviors.receiver()` node to the view-side sub-program. In our case, the following node is added:

```
1  const counter = Behaviors.receiver();
```

The transformation is depicted in Figure 6. As you can see, the croquetify function takes two inputs, the original single-user program and the "modelRealm" specification, and produces two sub-programs. Notice that the node definitions in the original program do not have to be modified. They are grouped into two programs, and some nodes are added.

The generated model and view code are "wrapped" in subclasses of `Croquet.Model` and `Croquet.View`, shown in Figure 7, respectively. One can think of the simple `count` property in Figure 3 as becoming a `ProgramState` with a few nodes including `counter` in `MyRenkonModel` in Figure 7. `MyRenkonModel` uses Croquet's `subscribe` API to register

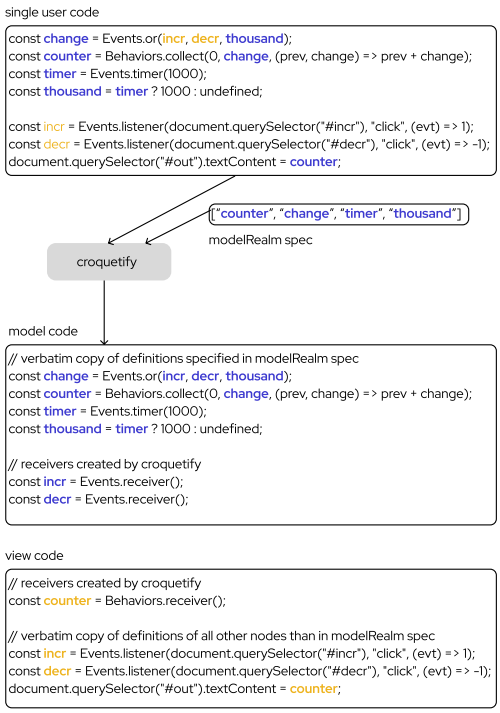

Figure 6. The croquetify function takes the original single-user program in Renkon and the model realm specification, and generates two Renkon programs. It analyzes the dependency relationships that cross the model-view boundary and generates `receiver()`s to connect them.

the handler called `viewMessage` when the view sends an event (lines 9–10). The `viewMessage` method (line 13) calls the `registerEvent` method (line 19) to inject a value for a `receiver()` into the instance of `ProgramState`, and then calls `evaluate` on it (line 25 via the method `run`). In effect, the nodes in the model code are updated according to the event sent from the view. After the evaluation cycle ends, the names of nodes that are updated and to be sent to the view are collected (line 25), and the model sends a `modelUpdate` event to the view (line 26).

On the `MyRenkonView` side, the "announcer" (line 53) publishes a Croquet event when a node value is updated and it is to be sent to the model. The `modelUpdate` method (lines 44–51) receives a set of names and injects their values into the instance of `ProgramState` at once (that is, for evaluation at the same logical time). The view-side `ProgramState` is configured to run automatically as needed, meaning that a call to `registerEvent` (line 48), or user interaction such as a button press, triggers an evaluation cycle.

Generating `MyRenkonModel` and `MyRenkonView` with croquetify is typically done in the browser, generating the program text and calling `eval` to create the necessary structure. A programming environment running in a browser tab may invoke croquetify frequently, each time the program is tested.

Why do we add `Events.receiver()`s to the model and `Behaviors.receiver()`s to the view? A Croquet view-to-model event is typically a user event that causes a change in the persistent model state. On the other hand, a notification from the model to the view is typically used to trigger screen updates, and the view reads all necessary quiescent data at once from the model. This difference maps very well to the FRP Events and Behaviors distinction, and it shows that the concepts from FRP and Croquet work together nicely.

In our example, when the user clicks on the `incr` button on the screen, the `incr` event gets a new value (+1), and an evaluation cycle of the view-side program is triggered. (In this simple program, there are no dependents on `incr` on the view side, but one could imagine some view-side visual effects being triggered.) During the evaluation cycle of the view dependency graph, each time a node referenced by the model gets a new value, a Croquet event is sent even while the view-side Renkon evaluator is traversing the node list. In other words, events from the view to the model are not "batched" but are sent as soon as the change on a node occurs.

The view-to-model event is sent via the reflector to the replicas of the model on all participants. The receiving model injects the event into the corresponding `Events.receiver()` and triggers an evaluation cycle. In this way, the events sent from different remote clients are sequenced into a unique order, and they trigger updates in the model in a bit-identical manner without interleaving the effects.

When the model sub-program is evaluated, there may be a number of nodes that get new values. We send only one Croquet model-to-view event that has all changed node names as its payload. The view-side Croquet event handler extracts values from the model (recall that the view can read the model data directly as long as it does not mutate it), injects the values into the corresponding `Behaviors.receiver()`s all at once, and runs an evaluation cycle.

Again, this transformation is simple, as the definitions of nodes are loosely coupled, connected only by the names of free variables. One can simply concatenate the definitions of nodes to create a new program.

When we transform an FRP program, we need to ensure that it does not introduce glitches. Recall that a glitch results from temporary inconsistent evaluation, as described in Section 2. If a transformation introduces dependencies that cannot be topologically sorted, there will be no consistent evaluation order for the nodes. While we have not provided a formal proof that this transformation does not introduce any glitches, it is easy to see why. A sketch of the proof is as follows: given a graph G that is acyclic and topologically sortable, we partition it into disjoint subgraphs G_{model} and G_{view}. Since the added receiver nodes are only sources, they introduce no new dependency edges, and therefore cannot invalidate the topological ordering.

```
class MyRenkonModel extends Croquet.Model {
  init() {
    this.programState = new ProgramState(0, this);
    this.programState.updateProgram([modelNodeStr]);
    this.programState.options = {once: true};

    this.programState.evaluate(this.now());

    this.subscribe(this.id, "viewMessage",
      this.viewMessage);
  }

  viewMessage(data) {
    const now = this.now();
    const {name, value} = data;

    if (name === undefined ||
        value === undefined) {return;}
    this.programState.registerEvent(name, value);

    this.run(now);
  }

  run(now) {
    let changedKeys = this.programState.evaluate(now);
    this.publish(this.id, "modelUpdate", changedKeys);
  }
}
```

```
class MyRenkonView extends Croquet.View {
  constructor(model) {
    ...
    this.programState = new ProgramState(0, this);
    this.programState.updateProgram([viewNodeStr]);
    this.programState.announcer = (varName, value) => {
      this.announcer(varName, value);
    };
    this.programState.evaluate(this.now());

    this.subscribe(this.model.id,
      {event: "modelUpdate", handling: "oncePerFrame"},
      this.modelUpdate);
  }

  modelUpdate(keys) {
    for (const key of keys) {
      const value = this.model.programState.resolved.get(key);
      if (value && value.value !== undefined) {
        this.programState.registerEvent(key, value.value);
      }
    }
  }

  announcer(varName, value) {
    if (this.model.viewToModel.has(varName)) {
      this.publish(this.model.id,
        "viewMessage", {name: varName, value: value});
    }
  }
}
```

Figure 7. The model and view code that wraps generated Renkon dependency graphs.

5 Croquetifying Renkon-pad

In the previous section, we introduced the croquetify function, which converts a single-user application written in Renkon to a multi-user one. In this section, we show that we can croquetify substantial programs.

Renkon-pad is an interactive live programming environment with a zoomable infinite 2D field in which to write a Renkon program (see Figure 1). The user can instantiate multiple code editors and edit Renkon node definitions in them. The user can create multiple "runners", which are `iframes` that run the Renkon program being edited. The receiving `iframe` creates or updates an instance of `ProgramState` and runs the program.

The code editor is an instance of the unmodified CodeMirror editor [12]. We could use any JavaScript-based text editor, but we find that CodeMirror has good support for editing JavaScript code, and it also has the official `collab` package to support multi-user editing. (We explain this in detail in Section 5.2.)

The version of Renkon-pad as of early 2026 has about 180 top-level node definitions and about 3,000 lines. It grew from an earlier version with about 700 lines that had reached the self-sustaining threshold, meaning that a programmer could edit its code in it and save the result. Renkon-pad is a practical environment. The authors use it for their own development work on a daily basis.

As described in the Data Structure section of [19], the application data of Renkon-pad is represented with only a handful of Behavior nodes. They are the `windows`, which is an array of window IDs; `positions`, which is a dictionary (Map) of rectangle objects (`{x, y, w, h}`, keyed by a window ID); `windowTitles`; etc. We can start with this handful of nodes in the `modelRealm` spec for croquetifying this program. It turns out that this is already good enough to make the window-manipulation part of the system multi-user.

> Note to readers:
>
> When you have multiple Renkon-pad browser tabs in the same session, you can press the "code" button at the top of one tab and drag the created window. The creation and movement will be synchronized across all tabs.

5.1 Remote Viewports and Cursors

For a multi-user programming environment with a zoomable interface to be useful and practical, it needs more features than just window manipulation. For one, users typically want to see what parts of the 2D field remote collaborators are

viewing and where their mouse pointers are (the remote viewport and remote cursor features). This means that the data in the Croquet model should include the position and extent of the viewport, for instance. The viewport is a dictionary of `{x, y, scale}` objects keyed by the `viewId`, which uniquely identifies a participating browser window in a Croquet session. Similarly, the cursor position in the logical field coordinates is also stored in the Croquet model, and the view renders it on the screen.

> Note to readers:
>
> You can zoom in and out and pan in one of your browser windows and see how the translucent rectangles change in other browser windows. Care is taken so that the remote cursor of a user is visible at the edge of your screen even when their viewport and yours do not overlap.

5.2 Multi-user Text Editors

Editing textual code (and documentation in Markdown) in those boxes is the central activity in using Renkon-pad, and it is essential to support simultaneous multi-user editing. We use CodeMirror's `collab` package in conjunction with Croquet. The `collab` package assumes that the "authority" of the collaboration receives a CodeMirror transaction with a "version number" that identifies the point in the linear progression of editor state. This version number is used by the authority to determine if transactions from different participants interleave; if so, it transforms a transaction. This works very well with Croquet because Croquet's reflector sequences the events; thus, the replicated Croquet models running on different clients can act as the authority of `collab`.

Following `collab`'s reference implementation, we have created a subclass of `Croquet.Model` called `CodeMirrorModel`. We create an instance of it for each text editor in a session (so potentially there may be hundreds of instances in the session). Each `CodeMirrorModel` contains a CodeMirror `Text` object. When it receives a transaction from a participant, it may transform it, apply the transaction to the `Text`, and send the transformed transaction to other participants. The participants' views apply that transaction to CodeMirror's `EditorView` to update the edit state on their own screens.

> Note to readers:
>
> You can start typing in a code editor window in one of the browser windows and see the code editor in other browser windows update. We have added multi-cursor support too; you can see remote users' insertion points in the text.

5.3 Synchronized Runner

The program that is being edited in a Renkon-pad environment is run in a separate execution context. Each runner window has an `iframe`, and when the runner's play button is pressed, the existing code in the environment is gathered and sent to the runner to run it. We want to replicate the play-button click so that when one of the users presses the play button, every participant starts the same program. This is done simply by introducing a Renkon node for handling the play button press into the model. This effectively synchronizes the play event for all participants.

5.4 The Size of the Renkon-pad Program

Renkon-pad has about 180 node definitions in slightly over 3,000 lines of Renkon. The model realm spec has 24 names, which include eight nodes that are essential application data (such as window positions and contents). There are additional nodes in the model to handle shared but transient state such as viewports, play button triggers, and file loading. The view has 155 nodes. It includes user interaction, rendering, text search, CSS definitions, and a CodeMirror plugin. Model/view cooperation requires just 15 model-to-view connections and 13 view-to-model connections. A large part of the line count is CSS and the construction of virtual DOM elements. There are user-interaction events (such as those for moving or resizing a window) that are sent from the view to the model, but the number of them is still small.

In addition to code written in Renkon, the definitions of `CodeMirrorModel` and `CodeMirrorView` are in a separate JavaScript file, which has about 500 lines.

There are very few external libraries used. The Renkon runtime uses the Acorn parser, and rendering is done with Preact and, of course, the browser. Text editors use CodeMirror, but everything else is written in Renkon.

5.5 Dynamic Switch Between Single- and Multi-user Modes

We want to keep the option of running Renkon-pad as a single-user app as well as a multi-user app from the same codebase. While the implementation of Renkon-pad almost "just works" for both cases, introducing `CodeMirrorModel` and `CodeMirrorView` has made the application depend on Croquet. To avoid linking Croquet statically to Renkon-pad, we added startup logic that determines whether the application is started in multi-user mode or not. We run croquetify at startup time only when the program is running in multi-user mode. When running in single-user mode, the definitions of `CodeMirrorModel` and `CodeMirrorView` are swapped with stub versions that create a CodeMirror instance directly.

Note to readers:

You can open https://tinlizzie.org/onward2026 without any URL parameters to run Renkon-pad as a single-user app. The same code with the `?q=` URL parameter and session password `pw=` works as a multi-user app.

There is one more axis in the choice between single- and multi-user modes besides the editor being in one of these modes. We also want to control whether the application being edited in the environment should itself be run as a single- or multi-user app. In the current implementation, it is controlled by the presence of a code editor with the reserved title "Croquet". When the environment contains a text editor with "Croquet", the content is sent to a runner separately. The content of the Croquet window instructs the receiving runner whether to run croquetify or not.

6 Related Work

We discuss related work to establish the contemporary and historical context for our project.

We trace the lineage of this work to TBAG by Elliott et al. [10]. TBAG was a networked collaborative framework that used a constraint solver on each client. An event is delivered to other clients, and each of them computes the new state. Analogously, we use FRP (another invention by Elliott et al.), which can be seen as a propagation-based one-way constraint solver. The Croquet network architecture of our work is more sophisticated and practical, as Croquet is well tested in products and examples, and some mechanisms such as snapshots (and persistent data) support long-running applications.

There have been numerous attempts to apply reactive languages to distributed applications [5][15]. A recent example is Meerkat by Costa Seco et al. [7]. Meerkat's goal is similar to ours, including support for live programming. However, our approach with Croquet and Renkon differs from theirs in some crucial respects. Renkon on Croquet allows us to develop an application starting as a single-user program and then use it as the starting point for a multi-user program. The developer can gradually enhance it with multi-user features. In other frameworks, this enhancement often breaks the initial assumptions about the program structure. In our scheme, starting with a clean, logical-time-aware language helps structure the program in a way that is conducive to incremental improvements.

Compared to an early attempt by Reynders et al. to write a network application as a "multi-tier FRP program" by splitting one program into multiple parts that run on the server and the local computer [23], the benefit of our approach is that it avoids writing a server-side program altogether. Avoiding application-specific server-side code alleviates much of the difficulty of developing client-server programs. The lightweight communication protocol of Croquet is also conducive to developing a real-time action game that requires a 60 Hz or higher event rate.

Newspeak-on-Croquet by Bracha is an attempt to use Croquet as a network transport mechanism to add collaboration features to the malleable programming environment Newspeak [4]. The goal is very similar to ours. However, the attempt does not embrace Croquet's model-view separation and propagates values across the network, which limits its ability to ensure the correctness of the synchronized application state.

In terms of (single-user) reactive language constructs, Signals [8] and Svelte's Runes [25] try to provide a way to describe dependency relationships among values in the program. Using Signals is cumbersome. One has to write:

```
1 const count = useSignal(0);
2 const double = useComputed(() => count.value*2);
```

to have reactive values called `count` and `double`, where `double` is `count * 2`. Svelte's runes are slightly simpler syntactically, but they still require explicit marks (`$derived` and `$effect`). Having to specify which signal is the base signal and which signals are derived or computed makes incremental improvement harder, and is also less conducive to program transformation than Renkon.

As a historical note, the property of Renkon, and of reactive programming in general, whereby the order of statements in a program does not affect execution, but only the dependency relationship does, can be traced to the Compel language by Tesler and Enea in 1968 [26].

6.1 Merge-based Solution vs. Croquet

Croquet's approach also differs from "merge-based" collaboration libraries such as Automerge and Y.js [3][13], which typically use CRDTs [24]. With such libraries, multiple clients make changes to a shared document simultaneously, potentially without communicating for a long time. The divergent data on clients are merged to create a single "truth". This approach, often called "local-first", has its benefits, as it allows offline editing [14]. However, a merge-based approach does not necessarily guarantee semantically correct results, not only when the changes are too large to be reasonably reconciled automatically but also when clients are online. Imagine a collaborative graphical shape editor where users create rectangles of different sizes on the screen, and they can move and resize them. The developer might later wish to impose a semantic restriction that all rectangles should be within the bounds of the owning canvas. The developer would tweak the local UI so that the move and resize handlers clip user pointer movement that would put the edge of the box out of bounds. A problem arises when user A moves a rectangle toward the right while user B resizes it to be wider toward the right on a remote computer. The merged result would make the rectangle stick out of bounds.

In Croquet, under the "network-first" assumption, the events are linearized, and these edit events can be rejected or transformed when a later one violates the application contract.

We point out that the combination of Renkon and Croquet offers a solution for one of the local-first ideals: "the network is optional". Because a Renkon app can work as both a multi-user and a single-user app, it allows the user to work on their document even when they are offline. For instance, the program developed in a collaborative session can be saved to the user's local disk. The user can then go offline, launch single-user Renkon-pad served from localhost, and load the saved program to continue developing. Later, the user's work can be merged with others' work via Git.

6.2 Croquet Architecture Limitations

We recognize that the network-first assumption imposes certain limitations. Croquet is not a fully decentralized architecture and requires a reflector that is reachable by all participants. A mitigating factor in the Croquet architecture is that the reflector does not store application state (except for a small number of events received since the last snapshot was taken), and there may be any number of reflectors deployed across the global network. This means that if the reflector serving a session goes down, the session can be migrated to another working reflector with the loss of at most a few seconds' events. The commercial versions of the Croquet network included brokers that took care of this migration. Keeping the network-side entity simple also has practical benefits for robustness. The commercial Croquet network typically achieved years of continuous service availability. Furthermore, as described in Section 3, the reflector and the Croquet client library address common issues such as WiFi disruptions, so reconnecting to an existing session works well.

In short, although the network-first assumption may appear to be a strict limitation, network availability is already a requirement for real-time collaborative applications (such as real-time games). Under that assumption, we believe that the Croquet network architecture provides a practical and robust foundation.

The Croquet architecture requires extra effort to run an application that depends on some server-side logic. For example, an application might involve large data, say, a dataset over 10GB. For such an application, we can run a "headless" client on a machine that has the dataset, and the custom view of the headless client can handle heavy computation. Processed results of much smaller size can be sent to regular clients. In other words, the Croquet architecture has different trade-offs from a typical client-server model, but a headless client with custom view code can support such cases.

This does not preclude research into a "best of both worlds" approach, namely a system that seamlessly supports both network-first and local-first operation.

7 Discussion and Future Work

We first summarize the "essence" of the work, and then discuss future work in this section.

7.1 Distilled Architectural Ingredients

From our experience, we identify the following design "ingredients" as being particularly important:

- Explicit logical time
- No application-specific server-side code
- Rich meta-programming features
- Loosely coupled program elements
- Distinction between triggers and persistent application state (namely, Events and Behaviors)

Explicit logical time and the distinction between triggers and persistent application state (or Events and Behaviors) help ensure that the application state is computed from consistent values, even when the network is involved. Eliminating application-specific server-side code, combined with loose coupling, allows programs to be modified dynamically without requiring any special support from the server. The rich meta-programming features allow programs to be treated as data and sent over the network without dealing with closures or other objects referenced by functions. Other systems and languages that share similar characteristics could adopt the proposed technique.

We also contend that providing a robust foundation for collaboration together with live programming capability requires a ground-up system design. We posit that the combination of Renkon and Croquet works well in practice because both systems were designed to support this style of interactive development over a network. The combination takes advantage of the fact that both systems were designed with these goals in mind. In other words, the above list by itself cannot be applied to other languages and systems without careful design to create a robust collaborative system.

7.2 Possible Future Improvements

There are some mismatches between Croquet and Renkon. A Renkon program often uses JavaScript functions as helpers. In the single-user case, such a function is treated as the constant value of a stream. However, Croquet cannot serialize a function object as state (due to JavaScript limitations); thus, the model sub-program cannot have a function as a value. A Croquet program written in an object-oriented style can define a method for this purpose, as the actual model definition itself does not have to be serialized.

The croquetify function in its current form actually takes a set of function strings to be inserted into the generated class (such as `MyRenkonModel` in Figure 7). This works fine but often leads to duplicated code. Finding a more seamless integration is an area for further exploration.

A Renkon program can be written in TypeScript syntax, and in the authors' experience, the regular TypeScript typing rules are mostly sufficient. However, there is room for improvement to capture a class of incorrect programs. For example, when a reaction to an event calls `Events.send`, the value is only available in the next evaluation cycle. On the other hand, the value returned from the reaction is available only in the current evaluation cycle. A common error is to write a reaction that depends on both of these values, but it typically never fires. A type rule could catch such an issue when a reaction that calls `Events.send` is used in other ways.

A limitation of croquetify is that the Croquet model and messages from the view to the model need to be serializable. A single-user version of an app may use non-serializable objects, such as DOM elements, to store some essential information. For the croquetified program to work, the developer must change it so that the model stores only the essential information. This is typically done simply by separating the node into two nodes.

The Renkon language handles only one-way constraints as a deliberate choice. While this works for a large class of programs, some concepts, for instance graphical object layout, are inherently multi-way. A Renkon program typically offloads layout to the CSS engine, which performs its layout calculation after the last steps of an evaluation cycle, and avoids depending on its results. This works for typical cases, but if a program indeed needs to use the actual elements' locations, it often becomes difficult. One could imagine supporting multi-way constraints or adapting a language with a "stratified" evaluation scheme such as the Dedalus language [1]. The croquetify process could be applied to such a language.

Supporting programming with AI coding agents is a very important topic. As of this writing, trying to create a robust collaborative app does not work reliably due to the lack of examples and insufficient knowledge of how to handle corner cases. AI may produce an application that works when the developer casually tests it alone with a keyboard and a mouse, but it often breaks when two or more users actually interact with it in real time. Creating a "skill" for agents [2] that teaches best practices for Croquet and Renkon will be very valuable.

8 Conclusion

We set out to find a way to write collaborative applications more cleanly and easily. The "holy grail" that developers have been looking for in making collaborative applications is a programming language or system where the developer writes code for an application as if it is single-user, but it works as a multi-user app. We think the combination of Renkon and Croquet shows a lot of promise in that direction, for two main reasons, though it may not be "it" yet.

The characteristics of Croquet and Renkon help us achieve the goal. Both Croquet and Renkon use explicit logical time to advance the application state. This eliminates certain classes of race conditions and unintended breakage of application contracts.

The basic formulation of Croquet is simple, but careful and extensive engineering efforts devoted to it have made it a practical foundation for building real-time collaborative applications.

Renkon provides a clean way to describe dataflow, and the loose coupling of nodes in the dataflow graph allows us to transform a program easily. The combination yields a clean way to write an application and transform its code to be multi-user.

We have shown small and large examples. The counter program is written in a few lines. The large example, the collaborative programming environment, is about 3,000 lines, with features such as remote cursors and collaborative text editing.

We also provided a list of key ingredients that we believe helped make this experiment successful and could possibly be applied to other languages and systems.

Acknowledgments

The authors would like to thank the reviewers for their valuable suggestions. Ted Kaehler, Alan Kay, John Maloney, Nikolai Suslov, Adam Bouhenguel, and John Underkoffler provided comments on the draft. We would like to thank the late Andreas Raab, who worked on creating the Smalltalk version of Croquet, and also the late Vanessa Freudenberg, who spearheaded the design and implementation of the JavaScript version of Croquet.

References

[1] Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. 2010. Dedalus: Datalog in Time and Space. In *Proceedings of the First International Conference on Datalog Reloaded* (Oxford, UK) *(Datalog'10)*. Springer-Verlag, Berlin, Heidelberg, 262–281. doi:10.1007/978-3-642-24206-9_16

[2] Anthropic. 2025. Extend Claude with skills. https://code.claude.com/docs/en/skills.

[3] Automerge contributors. circa. 2017. Automerge. https://automerge.org/.

[4] Gilad Bracha. 2025. Online Collaboration for Free. https://newspeaklanguage.org/pubs/newspeak_on_croquet.pdf.

[5] Adam Chlipala. 2015. Ur/Web: A Simple Model for Programming the Web. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 153–165. doi:10.1145/2676726.2677004

[6] Preact contributors. 2015. Preact. https://preactjs.com/.

[7] João Costa Seco and Jonathan Aldrich. 2024. The Meerkat Vision: Language Support for Live, Scalable, Reactive Web Apps *(Onward! '24)*. Association for Computing Machinery, 54–67. doi:10.1145/3689492.3690048

[8] ECMA Technical Committee 39. 2023. JavaScript Signals standard proposal. https://github.com/tc39/proposal-signals.

[9] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming* (Amsterdam, The Netherlands) *(ICFP '97)*. Association for Computing Machinery, New York, NY, USA, 263–273. doi:10.1145/258948.258973

[10] Conal Elliott, Greg Schechter, Ricky Yeung, and Salim S. Abi-Ezzi. 1994. TBAG: a High Level Framework for Interactive, Animated 3D Graphics Applications. In *Proceedings of the 21th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1994, Orlando, FL, USA, July 24-29, 1994*, Dino Schweitzer, Andrew S. Glassner, and Mike Keeler (Eds.). ACM, New York, NY, USA, 421–434. doi:10.1145/192161.192276

[11] Vanessa Freudenberg. 2020. Croquet: A Unique Collaboration Architecture. Video is available at: https://www.youtube.com/watch?v=ujOVHVAjXj4.

[12] Marijn Haverbeke and contributors. 2007. CodeMirror. https://codemirror.net/.

[13] Kevin Jahns and contributors. [n. d.]. Y.js. https://yjs.dev/.

[14] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 2019. Local-first software: you own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Athens, Greece) *(Onward! 2019)*. Association for Computing Machinery, 154–178. doi:10.1145/3359591.3359737

[15] Geoffrey Litt, Nicholas Schiefer, Johannes Schickling, and Daniel Jackson. 2023. Riffle: Reactive Relational State for Local-First Applications. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) *(UIST '23)*. Association for Computing Machinery, New York, NY, USA, Article 76, 16 pages. doi:10.1145/3586183.3606801

[16] Marijn Haverbeke and contributors. 2012. Acorn Parser. https://github.com/acornjs/acorn.

[17] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: a Programming Language for Ajax Applications. *SIGPLAN Not.* 44, 10 (Oct. 2009), 1–20. doi:10.1145/1639949.1640091

[18] Yoshiki Ohshima. 2024. Renkon-Core: An FRP Evaluator. https://github.com/yoshikiohshima/renkon.

[19] Yoshiki Ohshima, Adam Bouhenguel, and Matthew Good. 2025. Renkon-Pad: A Live and Self-Sustaining Programming Environment Based on Functional Reactive Programming. In *Companion Proceedings of the 9th International Conference on the Art, Science, and Engineering of Programming (Programming 2025) (Open Access Series in Informatics (OASIcs), Vol. 134)*, Jonathan Edwards, Roly Perera, and Tomas Petricek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 11:1–11:20. doi:10.4230/OASIcs.Programming.2025.11

[20] Yoshiki Ohshima, Aran Lunzer, Vanessa Freudenberg, Brian Upton, and David Smith. 2022. Live Programming and Text Editor Integration in the Croquet Microverse 3D Collaborative Construction System. *LIVE 22*.

[21] Marc Pouzet. 2006. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI.

[22] React. 2013. React. https://react.dev/.

[23] Bob Reynders, Dominique Devriese, and Frank Piessens. 2014. Multi-Tier Functional Reactive Programming for the Web. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Portland, Oregon, USA) *(Onward! 2014)*. Association for Computing Machinery, 55–68. doi:10.1145/2661136.2661140

[24] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems* (Grenoble, France) *(SSS'11)*. Springer-Verlag, Berlin, Heidelberg, 386––400.

[25] Svelte Runes. 2024. Svelte Runes. https://svelte.dev/docs/svelte/what-are-runes.

[26] L. G. Tesler and H. J. Enea. 1968. A Language Design for Concurrent Processes. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference* (Atlantic City, New Jersey) *(AFIPS '68 (Spring))*. Association for Computing Machinery, New York, NY, USA, 403–408. doi:10.1145/1468075.1468134

[27] The Croquet Team. 2017. The implementation of the Croquet library and the reflector. https://github.com/croquet/croquet.

Received 2026-05-15; accepted 2026-06-23