Symmetric Peer-to-Peer Applications

Francisco Sant'Anna *Department of Computer Science, Rio de Janeiro State University*

Abstract—In multi-node collaborative applications, such as shared online documents, users can interact remotely and yet share the exact same experience as if they were in a single machine. In this work, we propose a middleware for *symmetric peer-to-peer applications* in which decentralized instances can broadcast asynchronous events and yet conform to identical behavior. Nodes are allowed to join and leave the network at any time. Application developers must adhere to a restricted API, which is deterministic, stateless, and only supports pre-allocated memory. The middleware is responsible for synchronizing the events in a global shared timeline across the network. Based on memory snapshots and deterministic simulation, a "time machine" can rollback conflicting peers to resynchronize their state. TODO: application, results

Index Terms—collaboration, determinism, peer-to-peer, time machine

1 Introduction

Collaborative networked applications allow multiple users to interact remotely and yet share the same experience in real time. Examples of these *symmetric distributed applications* [1] are shared documents, watch parties, and multi-player games. In order to reproduce the exact behavior in multiple devices, the application must be able to synchronize time and execution across the network. In addition, since users can interact asynchronously with the application, event occurrences must somehow be synchronized back across devices.

A common approach towards symmetric applications is to introduce a middleware to orchestrate events and time in the network [1], [2]. This way, applications developers can rely on middleware primitives to trigger events, which are intercepted and synchronized in a global shared timeline across the network. Developers must also restrict themselves to deterministic and stateless calls only, such that execution can be equally reproduced in all nodes according to the shared timeline. However, current solutions depend on a central server to interconnect network nodes and determine the shared timeline.

In this work, we propose *symmetric peer-to-peer applications*, which do not rely on central servers for coordination. Nodes in the application form a dynamic graph network and communicate only with direct neighbours. Events are flooded in the graph and are triggered locally with a small delay to consider the network latency. To deal with events received out of order or too late, a distributed time machine can rollback peers to a previous state and reapply events in order and in time.

TODO: application, results

In Section 2, we revisit existing solutions for symmetric distributed applications. In Section 3, we detail our proposed middleware. In Section 4, ... In Section 5, ... In Section 6, ...

- assumptions - correct (non-malicious) nodes

- local first - p2p - less restrictive (compared to CRDTs) - semantic events - limitation b/c of rollbacks - features b/c of network traffic

	Network	Timeline	Events
Croquet	centralized	sync	online
GALS	centralized	async	online
CRDTs	decentralized	async	offline
This Work	decentralized	async	both

Fig. 1.

2 Related Work

In this section, we revisit existing solutions for symmetric distributed applications. We focus on (a) how the network is organized, (b) how global time is determined, and (c) how events are propagated and applied. Figure 1 compares selected works regarding these aspects.

Croquet[1] [2] guarantees bit-identical real-time behavior for every user in a collaborative distributed environment. As detailed in Figure 2, a centralized *reflector* issues periodic ticks, such that all nodes in the network advance together according to a synchronized global shared clock. If an user generates an event, it is delayed and sent back to the reflector, which broadcasts the event in the next tick, which all nodes apply in sync (including the originating node). Croquet takes periodic snapshots of the whole application state in order to support late joins to a running session. The new node just needs to restore the latest snapshot and simulate the remaining events to reach the current running state. As indicated in Figure 1, Croquet relies on a centralized network, in which nodes advance in sync, and in which event outcomes depend on the central server to be online.

GALS [1] shares the same goals with Croquet, but with some tradeoffs, mostly favoring clients with slow connections. As detailed in Figure 2, instead of advancing ticks in sync with the server, clients have their own local clocks. Also, event generation is delayed dynamically according to the slowest client. On the one hand, ticks do not generate any traffic and clients have smooth frame transitions, even

1. Croquet.io: https://croquet.io/

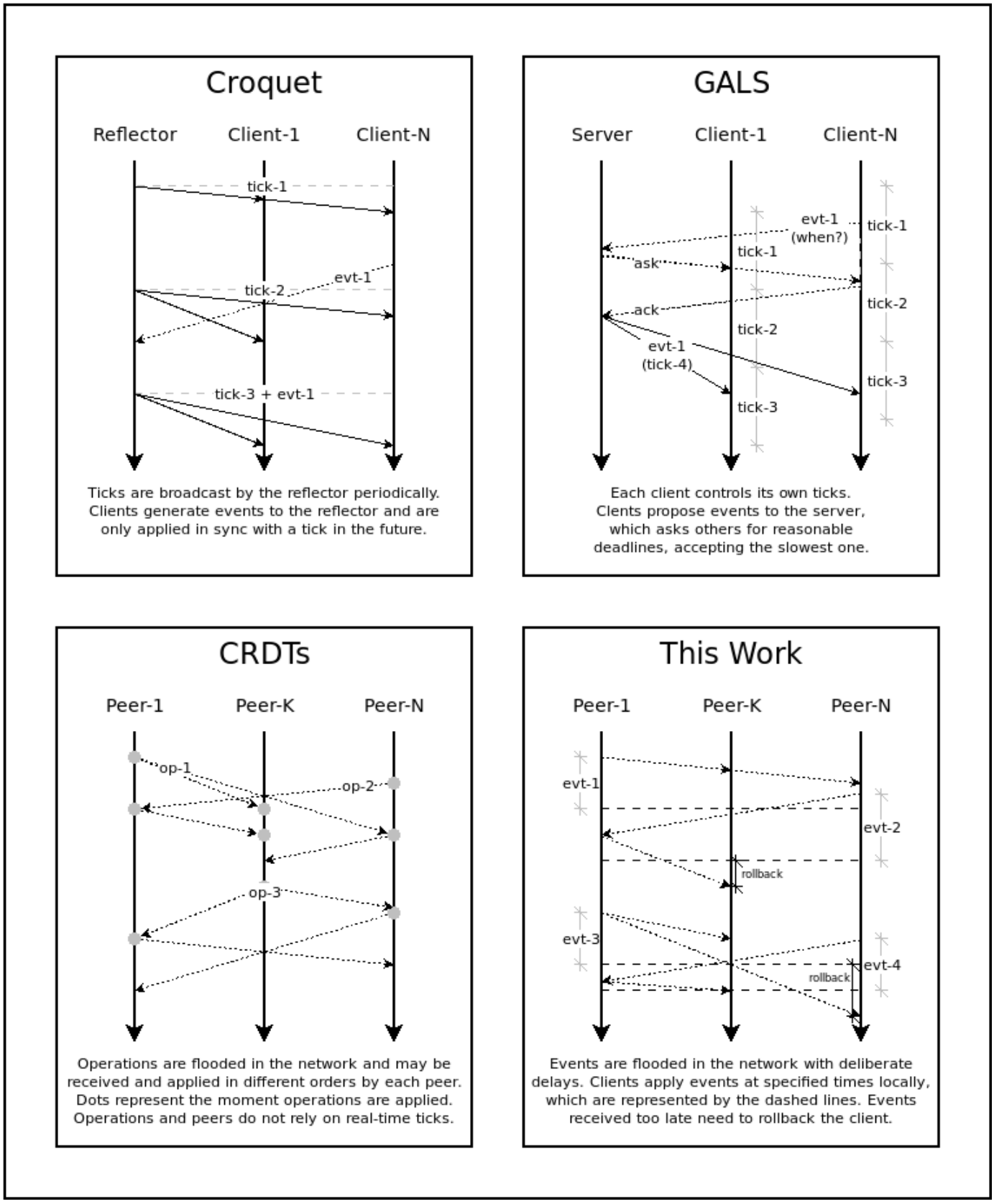

Fig. 2. Four approaches to symmetric distributed applications: Croquet, GALS, CRDTs, and this work. The vertical arrows in parallel represent the timelines in nodes in the network. The arrows crossing nodes represent communication between them.

those with poor connections. On the other hand, event generation requires two round trips to the server and clients may experience occasional freezes. The syncing protocol also requires bookkeeping to deal with clock drifts and client disconnections [1]. As indicated in Figure 1, GALS relies on a centralized network, in which nodes advance independently, and in which event outcomes depend on the central server to be online.

In both solutions, the network advances together as a whole in real time, with a total order among events, which is determined by a central server that must be permanently online. All clients compute the events in sequence, respecting their timestamps, and using only deterministic and stateless calls. This way, it is guaranteed that all clients reach the same state, also going through bit-identical steps.

An antagonistic approach to deterministic reactions to events is to model the application with conflict-free replicated data types (CRDTs) [3]. CRDTs are designed in such a way that all operations are commutative, so that the order in which they are applied does not affect the final outcome. As detailed in Figure 2, peers can communicate operations directly to each other with no central authority. Also, since operations need not to be ordered, they can be applied at the very first moment they are generated or received, even if the peers are offline. Since operations must be commutative, it is not possible to timestamp them in a global shared clock. Hence, there is no notion of a timeline in which peers go through bit-identical steps, but they do eventually reach the same state. On the one hand, the main advantage of CRDTs is the support for local-first software [4], since they can work offline in the same way as online. On the other hand, they provide very restrictive APIs (the CRDTs themselves), and do not support real-time consensus among peers. As indicated in Figure 1, CRDTs work on decentralized networks, in which peers advance independently, and in which operations can be applied immediately, even while offline. Automerge along with its accompanying Hypermerge peer-to-peer infrastructure is an example of an implementation of decentralized CRDTs [5], [6].

In this work, as indicated in Figure 1, our goal is to support peer-to-peer real-time symmetric execution, while still tolerating offline and out-of-order event generation. As detailed in Figure 2, peers have independent timelines and can communicate events directly to each other. Events are delayed to be able to reach other peers in time. In the case a deadline is missed, the peer rolls back in time and then fast forwards execution until it catches real-time behavior again. Like Croquet and GALS, peers compute events in sequence using only deterministic and stateless calls.

Regarding decentralized time machines, Fusion[2] is a state synchronization networking library for the Unity game engine. It provides networked tick-based simulation in two modes: client-server (*hosted mode*) and peer-to-peer (*shared mode*). The hosted mode is based on continuous memory snapshots, which allows for full rollbacks when clients diverge. However, the shared mode is less powerful and only supports eventual consistency among clients. We presume that continuous memory snapshots would be too costly without a central server.

2. Fusion: https://doc.photonengine.com/en-us/fusion

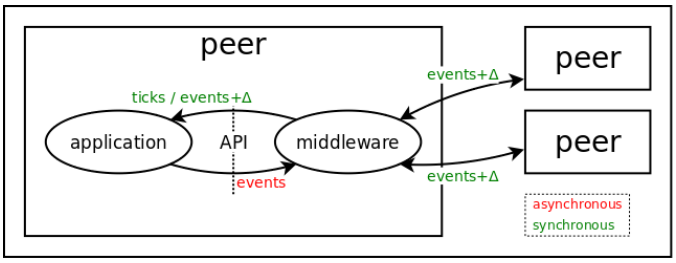

Fig. 3. Applications generate asynchronous events and communicate through an API with the middleware. The middleware orchestrates the peers, synchronizing their ticks locally and broadcasting events with a delay.

Regarding single-user time machines, the video game Braid[3] is designed on the assumption that players have unrestricted rollbacks as part of the game mechanics. Like Fusion, the implementation is based on continuous memory snapshots, instead of deterministic simulation as we propose in this work. Schuster [7] proposes a simple API for time travelling in event-driven JavaScript: `init()` to return an initial state; `handle(evt,mem)` to process events and modify state; and `render(mem)` to output the current state. As we describe next, we use a similar approach by separating mutation from rendering [8].

3 The Middleware & Programming API

In this section, we describe the architecture of our middleware and also the API in which developers have to conform with. The middleware is written in C and relies on the library *SDL*[4] to deal with the network and input events, such as timers, mouse clicks and key presses. The full source code[5] is under 500 lines of code.

We guide our discussion through the didactic example of Figure 4, in which multiple users use the arrow keys to control the direction of a moving rectangle. The six instances are connected in a peer-to-peer network, such that some peers can only communicate indirectly. The application is symmetric in the sense that if any peer presses a key, a corresponding event is broadcast and applied in all peers simultaneously, as if they were a single mirrored instance. At any moment, if a peer pauses the application, all peers are guaranteed to pause exactly at the same frame with the rectangle being "bit-identically" at the same position.

In Figure 3, a peer is depicted as an application that uses an API to communicate with the middleware. The application itself is not aware of the network and has no direct access to the other peers, which communicate transparently through the middleware. The application generates asynchronous events that go through the middleware, which synchronizes them with a timestamp in the future, such that all peers can agree. The middleware controls the execution of the application by issuing ticks and synchronized events.

The main job of the middleware is to deal with the uncertainties of the network in order to preserve the symmetric behavior across peers. More specifically, the middleware

3. Braid:http://braid-game.com/
4. SDL: www.libsdl.org
5. TODO

ensures that all peers (a) meet event deadlines, (b) advance in sync in real time, and (c) can leave and join the network and remain symmetric:

a) **Event Deadlines:** Due to the inherent latency of networks, peers generate events with a deliberate delay so that they reach other peers in time to be applied in sync in a common deadline. However, a peer that is locally ahead of time may receive an event that should have been applied in the past, event considering the delay. In this case, the middleware automatically rolls back the peer, applies the event at the correct time, and then fast forwards the peer, re-applying all remaining events up to the real time. As we detail further, rolling back a peer requires to simulate its whole execution from the beginning (or from a snapshot in between).

b) **Time Synchronization:** Since peers run in different machines, their timelines will inevitably be out of sync because: (a) applications are launched locally at different times, and (b) internal clocks may diverge over time. The middleware assumes the maximum time among all peers the be the "correct real time", so that other peers must fast forward to catch up with it. Therefore, in order to determine the maximum time and synchronize the clocks, the middleware proceeds as follows: (a) peers broadcast `SYNC` events periodically with their local times, and (b) if a peer is behind a received `SYNC`, it advances its frames proportionally faster to catch up in time.

c) **Peer Churn:** Considering that we target peer-to-peer networks, it is important that the middleware can handle arrival and departure of peers seamlessly. When a peer leaves the network, nothing needs to be done. The middleware just needs to ensure that a peer that joins the network receives all events in order, which is trivial since they all hold an absolute timestamp.

3.1 The Programming API

The middleware expects to take full control of the application execution in order to control its timeline and disseminate events in the network. For this reason, the basic programming API is to call a `loop` function passing a set of callbacks, which the middleware uses to communicate with the application:

```
int main (void) {
    p2p_loop (
        1,              // peer identifier
        50,             // simulation FPS
        sizeof(G), &G,  // pre-allocated full state
        cb_ini,         // init/quit callback
        cb_sim,         // simulation callback
        cb_ren,         // rendering callback
        cb_evt          // events callback
    );
}
```

The middleware assumes that each peer is assigned a unique identifier in a contiguous range. The FPS must be the same in all peers to ensure that the simulation executes exactly the same.

3.1.1 Application State

The global variable `G` must hold the full application state. If dynamic allocation is required, it is necessary to hold a finite memory pool in `G` with a custom allocator. For instance, the moving rectangle application only requires to hold the current position and direction:

```
struct {
    int x,  y;   // position
    int dx, dy;  // direction speed
} G;
```

3.1.2 Application Events

The application generates events that need to be broadcast to the network so that all peers behave the same. In this sense, application events may differ from low-level local system events, which may or may not be mapped 1-to-1 in the callback `cb_evt`.

There middleware predefines two events for all applications: `P2P_EVT_START` represents the beginning of the simulation, and `P2P_EVT_TICK` called at every frame. The application-specific events must be declared by the programmer in an enumeration. In our application, we define a new event `P2P_EVT_KEY` to represent key presses:

```
enum {
    P2P_EVT_KEY = P2P_EVT_NEXT
};
```

3.1.3 Initialization Callback

The middleware calls `cb_ini` twice: at the beginning and at the end of the loop. It should be used to initialize (and finalize) the network topology, as well as static globals that can live outside the simulation memory, such as the window and image textures:

```
void cb_ini (int ini) {
    if (ini) {
        <...> // create the SDL window
        <...> // open the arrow PNG images
        // create peer links (peer-id, ip, port)
        p2p_link(2, "192.168.1.2", 5000);
    } else {
        <...> // destroy the SDL window
        <...> // destroy the arrow PNG images
        // destroy peer links
        p2p_unlink(2);
    }
}
```

3.1.4 Simulation Callback

The middleware calls `cb_sim` on every frame, passing an application event, to affect the simulation state `G` deterministically. The callback must never perform any side effects, such as stateful calls or rendering frames. In our example, we modify the state of the rectangle as follows: (a) reset its position and speed on `START`, (b) increment its position based on the current speed on `TICK`, and (c) modify its speed based on `KEY`:

```
void cb_sim (p2p_evt evt) {
    switch (evt.id) {
        case P2P_EVT_START: // resets pos and speed
            G.x = G.y = G.dx = G.dy = 0;
            break;
        case P2P_EVT_TICK:  // increases position
            G.x += G.dx * VEL;
            G.y += G.dy * VEL;
            break;
```

```
        case P2P_EVT_KEY:  // modifies speed
            switch (evt.pay.i1) {
                case SDLK_UP:
                    G.dx = 0;
                    G.dy = -1;
                    break;
                <...> // same for other keys
            }
            break;
    }
}
```

The calls to `cb_sim` normally happen during real-time simulation, i.e., while the user is interacting with the application. However, if the peer is out of sync, the middleware may "time travel" and call the simulation many times to go back and forth and catch up with real time. This is the reason why we define `START` as an event like any other, since travelling requires to simulate the execution from the beginning, event by event.

3.1.5 Rendering Callback

Time travelling is also the reason why `cb_sim` must be split from the rendering callback `cb_ren` [7], which would otherwise render the screen multiple times while travelling. In our example, `cb_ren` just needs to redraw the rectangle at the current position:

```
void cb_eff (void) {
    <...> // clears the SDL window
    // redraws the rectangle at the current position
    SDL_Rect r = { G.x, G.y, 20, 20 };
    SDL_DrawRect(&r);
}
```

3.1.6 Events Callback

The `cb_evt` callback is called by the middleware in real time, whenever a local SDL event occurs. This callback has two goals: (a) map and broadcast application events, and (b) provide instant feedback to the user. As mentioned previously, not all local low-level events may generate application events to be broadcast in the network. The callback is allowed to perform side effects, such as modifying the network topology, or exhibiting some feedback on the screen. In our example, we only generate application events for the 5 key events, and we also exhibit the pressed key in real time:

```
int cb_evt (SDL_Event* sdl, p2p_evt* evt) {
    if (sdl->type == SDL_KEYDOWN) {
        if (sdl->key == <any-of-the-keys>) {
            SDL_DrawImage(<img-of-the-key>);
            *evt = (p2p_evt){ P2P_EVT_KEY,sdl->key };
            return 1; // generate application event
        }
    }
    return 0; // otherwise, do not generate any event
}
```

3.2 TODO

TODO: simpler than vector clocks, absolute times are correct
TODO: piggyback SYNC in normal inputs

3.3 Middleware Orchestration

We now describe in detail how the middleware orchestrates an application and ensures that all peers (a) meet event deadlines, (b) advance in sync in real time, and (c) can leave and join the network and remain symmetric.

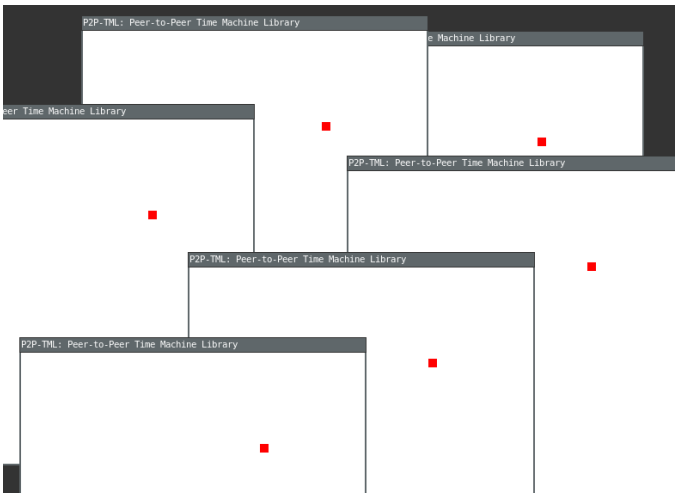

Fig. 4. A moving rectangle is controlled by peers connected indirectly. Users can press ↑ ↓ → ← to change the direction of the rectangle or `SPACE` to pause the application. The inputs are applied simultaneously in all peers, as if they were mirrors of a single application. Video: http://youtube.com/TODO

- p2p links por fora, como um servico TCP local - The network topology is dynamic

- time sync

The middleware handles: - time sync - disc / conn - late event

- walk back-forth - vector clocks - dynamic net - reconnect - late join

We make some assumptions about the network, which are assigned externally to the middleware:

- Peers have contiguous numeric unique identifiers.
- Peers identifiers are assigned externally to the middleware.
- Topology is dynamic, but also assigned externally

Except for the last item, we claim that the others can be solved externally, without affecting the middleware. For instance, - orthogonal and would not affect the other algorithms in the middleware

, which we claim that could be solved further: - peers have unique identifiers - ids are outside - topology, dynamic but outside - fixed maximum nodes - fixed latency We claim that Use external protocols For instance, latency is only a constant, no problem being a variable

- non-malicious nodes

4 Application

5 Evaluation

6 Conclusion

References

[1] F. Sant'Anna, R. Santos, and N. Rodriguez, "Symmetric distributed applications," in *Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, 2021, pp. 41–50.

[2] D. A. Smith, A. Kay, A. Raab, and D. P. Reed, "Croquet-a collaboration system architecture," in *First Conference on Creating, Connecting and Collaborating Through Computing, 2003. C5 2003. Proceedings.* IEEE, 2003, pp. 2–9.
[3] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Symposium on Self-Stabilizing Systems*. Springer, 2011, pp. 386–400.
[4] M. Kleppmann, A. Wiggins, P. van Hardenberg, and M. McGranaghan, "Local-first software: you own your data, in spite of the cloud," in *Proceedings of Onward'19*, 2019, pp. 154–178.
[5] M. Kleppmann and A. R. Beresford, "A conflict-free replicated json datatype," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 2733–2746, 2017.
[6] P. van Hardenberg and M. Kleppmann, "Pushpin: Towards production-quality peer-to-peer collaboration," in *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, 2020, pp. 1–10.
[7] C. Schuster and C. Flanagan, "Live programming for event-based languages," in *Proceedings of the 2015 Reactive and Event-based Languages and Systems Workshop, REBLS*, vol. 15, 2015.
[8] S. Burckhardt, M. Fahndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato, "It's alive! continuous feedback in ui programming," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 95–104.