Revista de Informática Teórica e Aplicada - RITA - ISSN 2175-2745
Vol. XX, Num. XX (2018) 11-XX

RESEARCH ARTICLE

Symmetric Peer-to-Peer Applications

Aplicações Peer-to-Peer Simétricas

Francisco Sant'Anna[1]*

Abstract: In real-time networked applications, such as shared 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 trigger asynchronous events and yet conform to identical behavior. Peers 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. Also, based on memory snapshots and deterministic simulation, a "time machine" can rollback conflicting peers to resynchronize their state. We show that the middleware can handle applications with over 20 peers in a heterogeneous topology under high churn. For instance, in a simulation scenario of 25 events per minute with a network latency of 50ms, the peers need to roll back and resynchronize only 3% of the time.
Keywords: collaboration — determinism — peer-to-peer — time machine

Resumo: Em aplicações de tempo real em rede, tais como documentos compartilhados, os usuários podem interagir remotamente e ainda obterem a mesma experiência, como se estivessem em uma única máquina. Neste trabalho, nós propomos um middleware para *aplicações peer-to-peer simétricas* nas quais instâncias descentralizadas podem disparar eventos assíncronos e ainda obterem comportamento idêntico. Os pares podem entrar e sair da rede a qualquer momento. Os desenvoledores de aplicações devem aderir a uma API restritiva, que deve ser determinística, sem estados internos, e também com alocação prévia de memória. O middleware é responsável por sincronizar os eventos em uma linha de tempo global e compartilhada em toda a rede. Além disso, com base em cópias da memória e simulação determinística, uma "máquina do tempo" pode retroceder pares em conflito para ressincronizar o seu estado. Nós mostramos que o middleware suporta aplicações com mais de 20 pares em uma topologia heterogênea e sob alta taxa de desconexões. Como exemplo, em uma simulação com 25 eventos por segundo em uma rede com latência de 50ms, os pares precisam retroceder e ressincronizar somente 3% do tempo.
Palavras-Chave: colaboração — determinismo — peer-to-peer — máquina do tempo

[1] *Department of Computer Science, Rio de Janeiro State University (UERJ), Brazil*
***Corresponding author**: francisco@ime.uerj.br
DOI: http://dx.doi.org/10.22456/2175-2745.XXXX • **Received:** dd/mm/yyyy • **Accepted:** dd/mm/yyyy

1. Introduction

Real-time networked applications allow multiple users to interact remotely and yet share the same experience. 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 a shared timeline.

In this work, we propose *symmetric peer-to-peer applications*, which do not rely on central servers for coordination. Peers in the application form a dynamic network graph and communicate only with direct neighbours, as in typical unstructured peer-to-peer networks [3]. Events are flooded in the graph and are triggered locally with a small delay to compensate the network latency. To deal with events received out

	(a) Network	(b) Timeline	(c) Events
Croquet	centralized	sync	online
GALS	centralized	async	online
CRDTs	decentralized	async	offline
This Work	decentralized	async	both

Figure 1. Related work regarding (a) how the network is organized, (b) how global time is determined, and (c) how events are propagated and applied.

of order or too late, a time machine can rollback peers locally to a previous state and reapply events in order and in time. Our main contribution is the design of a middleware to ensure 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. We target small peer-to-peer networks, in which nodes are only a few hops away from each other. This ensures that events can span the whole network in a reasonable time to preserve the real-time behavior of applications. We also assume that peers are non-malicious in the sense that they do not generate erroneous events.

We perform simulations with over 20 peers in a 5-hop mixed topology with cycles and straight lines. We vary the network latency and peer churn, as well as the frequency and deadlines of events. We then measure (a) the recurrence of time travels, (b) the real-time pace of peers, and (c) the recoverability from churns, such that they validate our goals above, respectively. As results, we show that the last two goals are met even under extreme conditions: peers exhibit a time mismatch of under 0.2% and take less than $1.5s$ to recover from churns. For the first goal, we identify the scenarios in which the middleware can meet event deadlines with the desired performance. As an example, in a scenario of 25 events per minute with a network latency of $50ms$, the peers need to roll back only 3% of the time.

In Section 2, we revisit existing solutions for symmetric distributed applications and software time machines. In Section 3, we detail the architecture of our middleware, its programming API, and how it orchestrates peers. In Section 4, we evaluate our design under a number of scenarios. In Section 5, we conclude this work.

2. Related Work

This section first revisits existing solutions for symmetric distributed applications, namely Croquet [2, 4], GALS [1], and CRDTs [5]. Then, we discuss software time machines that are built for either local or networked applications.

2.1 Symmetric Distributed Applications

Even though existing solutions aim to provide symmetric behavior for distributed applications, they diverge in three key aspects as follows:

a. How the network is organized.
b. How global time is determined.
c. How events are propagated and applied.

Figure 1 compares three selected works regarding these aspects, which we discuss and contrast with our approach next. At the end of this section, we also provide initial details on how our middleware works.

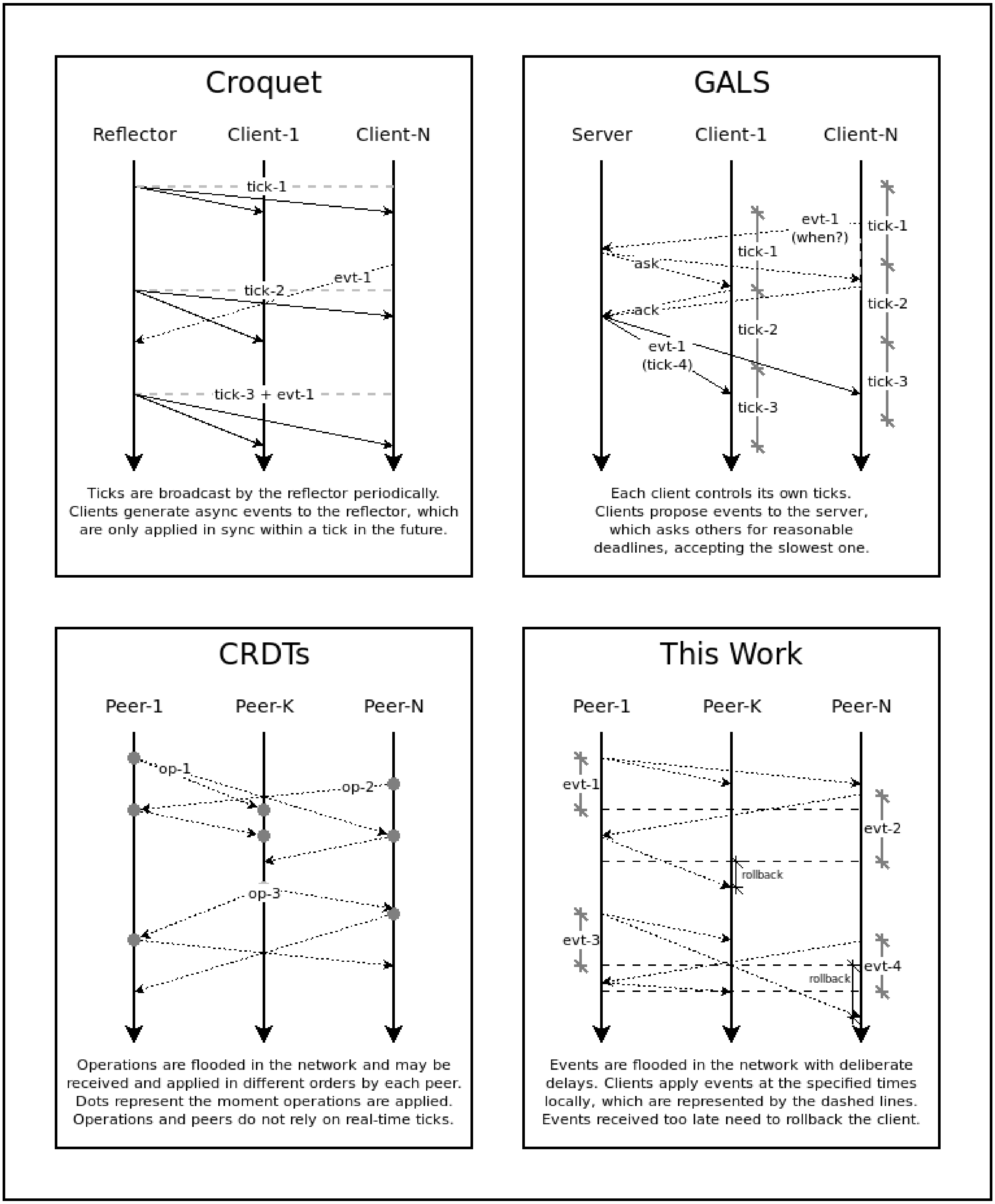
Croquet[1] [2] guarantees bit-identical real-time behavior for users in collaborative distributed environments. As detailed in Figure 2, a centralized *reflector* issues periodic ticks, such that all nodes in the network advance together according to a synchronized global shared clock. If a user generates an event, it is sent back to the reflector, which broadcasts the event in the next tick, which all nodes apply in sync. 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 (a), in which nodes advance in sync (b), and in which event outcomes depend on the central server to be online (c). Note how the central server is the only participant with knowledge about clients, which never communicate directly among themselves. This implies that the server must never fail, and also that periodic full broadcasts are the only possibility of spanning the whole network. In this work, we argue that in a peer-to-peer alternative any node can fail, and that broadcasts can be replaced by optimistic flooding.

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 independent local clocks. Event generation requires two round trips (*when* → *ask* → *ack* → *tick*) and 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 those with poor connections. On the other hand, events take longer to apply and clients may experience occasional freezes. The syncing protocol also requires extra bookkeeping to deal with clock drifts and client disconnections [1]. As indicated in Figure 1, GALS also relies on a centralized network (a), but in which nodes advance time independently (b), and in which event outcomes still depend on the central server to be online (c).

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 events in sequence, respecting timestamps, and using only deterministic and stateless calls. This way, it is guaranteed that all clients reach the same state, and observe bit-identical steps.

An antagonistic approach to deterministic reactions to events is to model the application with conflict-free replicated data types [5]. 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

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



Croquet

Ticks are broadcast by the reflector periodically. Clients generate async events to the reflector, which are only applied in sync within a tick in the future.

GALS

Each client controls its own ticks. Clients propose events to the server, which asks others for reasonable deadlines, accepting the slowest one.

CRDTs

Operations are flooded in the network and may be received and applied in different orders by each peer. Dots represent the moment operations are applied. Operations and peers do not rely on real-time ticks.

This Work

Events are flooded in the network with deliberate delays. Clients apply events at the specified times locally, which are represented by the dashed lines. Events received too late need to rollback the client.

Figure 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.

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 unique 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 to support local-first software [6], 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 (a), in which peers advance independently (b), and in which operations can be applied immediately, even while offline (c). Automerge along with its accompanying Hypermerge peer-to-peer infrastructure is an example of decentralized CRDTs [7, 8].

In this work, as indicated in Figure 1, our goal is to support peer-to-peer real-time symmetric execution (a), while still tolerating out-of-order (b) and offline (c) event generation. As detailed in Figure 2, peers have independent timelines and can communicate events directly to each other. Note, that it is not necessary that all peers communicate directly with all other peers, but only indirectly via unstructured flooding. 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 deterministic and stateless calls only.

Finally, note that classical distributed consensus mechanisms focus on fault tolerance [9], but not on real-time symmetric behavior. These algorithms typically rely on leaders [10], which require an election period that does not cope with our symmetric real-time requirements. In this sense, these algorithms could be considered to decentralize servers in Croquet or GALS, but not to decentralize symmetric peers with high churn rates, as we propose in this work. In contrast, our proposal requires a time machine to reapply reordered operations, since we cannot guarantee strong consistency [11].

2.2 Software Time Machines

A software time machine is a mechanism that allows to regress or advance the execution of an application. Time travelling is a established research topic for debugging tools [12]. In this context, it is possible to stop a program after a failure and revert its execution in order to understand the bug causes. In our context of real-time symmetric applications, time travelling is essential to re-synchronize the execution of mirrored instances that diverge due to asynchronous events.

Fundamentally, it is not possible to reverse the execution of most programs. The reason is because even trivial operations, such as ‘printf‘ and ‘abs‘ are not invertible. Therefore, time machines rely on techniques that either *record* the entire application states in a log, or *reconstruct* them from deterministic inputs [12]. In this work, we mix both techniques by taking periodic memory snapshots and simulating deterministic inputs in between them.

Regarding decentralized 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. As discussed in Section 2.1, this approach is similar to how Croquet handles late joins to a running session. The shared mode is less powerful and only supports eventual consistency among clients. We presume that the reason is because continuous memory snapshots would be too costly without a central server.

Regarding single-user local 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 [13] 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 state modification from rendering [14].

3. The Middleware & Programming API

The middleware is written in C and relies on the library *SDL*[4] to deal with the network, timers, and input events, such as mouse clicks and key presses. The full source code[5] is around 500 lines of code.

We guide our discussion through the didactic example of Figure 3, in which multiple users use the arrow keys to control the direction of a moving rectangle. The application is symmetric in the sense that if any user presses a key, a corresponding event propagates in the network, and all peers apply it simultaneously, as if they were a single mirrored machine. At any moment, if a user 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 4, 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 other peers, which communicate transparently through the middleware. The application generates asynchronous events that go through the middleware, which synchronizes them with a timestamp with a Δ delay scheduled to the future, such that all peers can (optimistically) satisfy. The middleware controls the execution of the application by issuing ticks and synchronized events.

The peers form a dynamic unstructured peer-to-peer network [3], and communicate mostly indirectly to each other.

[2] Fusion: ⟨https://doc.photonengine.com/en-us/fusion⟩
[3] Braid: ⟨http://braid-game.com/⟩
[4] SDL: ⟨http://www.libsdl.org⟩
[5] https://github.com/fsantanna/p2p-tml

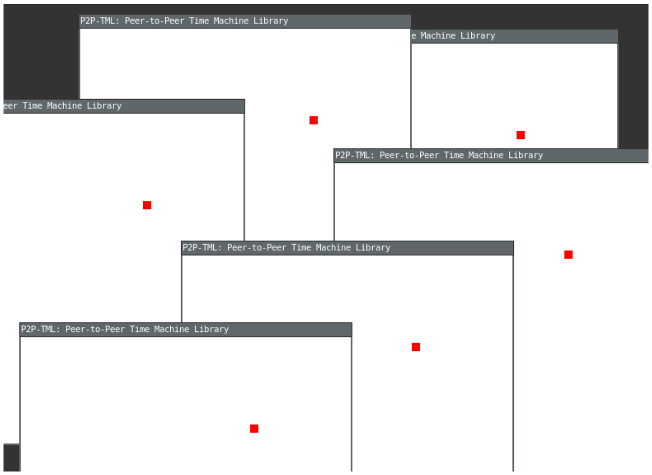

P2P-TML: Peer-to-Peer Time Machine Library
e Machine Library
eer Time Machine Library
P2P-TML: Peer-to-Peer Time Machine Library
P2P-TML: Peer-to-Peer Time Machine Library
P2P-TML: Peer-to-Peer Time Machine Library

Figure 3. 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: ⟨https://youtu.be/zkl0vSOGino⟩

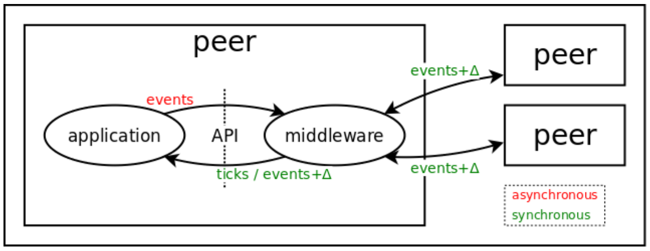


Figure 4. Applications generate asynchronous events and communicate through an API with the middleware. The middleware orchestrates the peers, synchronizing their ticks locally and disseminating events with a Δ delay.

Events are flooded in the network graph via gossiping, i.e.: when a peer generates an event, it communicates to its neighbours, which communicate to their neighbours, and so on. Note that all peers execute the exactly same application and middleware, with no differences with respect to their roles and physical resources. Therefore, we can describe our peer-to-peer network as follows: (a) all peers have the same role and run the same software; (b) any peer can join or leave at any time; (c) events are only communicated with direct neighbours.

The main job of the middleware is to deal with the uncertainties of the network in order to preserve the symmetric behavior across peers. As described in the Introduction, the middleware 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:

1. **Event Deadlines:** Due to the inherent latency of networks, peers generate events with a deliberate delta Δ so that they can reach other peers in time to be applied in sync. Nevertheless, a peer ahead of time may receive an event that should have been applied in the past, even 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, reapplying all remaining events up to the real time. As we detail further, rolling back a peer requires to simulate the whole application execution from the beginning. As an optimization, the middleware takes periodic snapshots locally at all peers to avoid full simulation.

2. **Time Synchronization:** Since peers run in different machines, their timelines will inevitably be out of sync because applications are launched locally at different times, and because internal clocks may diverge over time (e.g., from rollbacks or timer polling inaccuracies). The middleware assumes the maximum time among all peers to 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 disseminate 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.

3. **Peer Churn:** Considering that we target peer-to-peer networks, it is important that the middleware can handle arrival and departure of peers seamlessly. Note that high churn may lead to intermittent network partitions. Given our unstructured approach, nothing needs to be done when a peer leaves the network. However, when a peer joins the network, the middleware needs to ensure that it receives from other peer all events that ever happened, in order. This is trivial since all peers already need to hold the full event history (as described in item (a)), including their absolute timestamps in the shared

timeline. In addition, a peer that rejoins the network recovers from a past snapshot it already holds, thus avoiding full simulation.

3.1 The Programming API

The middleware expects to take full control of the application execution in order to manipulate 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 assigns itself a unique identifier in a contiguous range. The FPS must be the same in all peers to ensure bit-identical simulation.

3.1.1 Application State

The variable `G` passed to the middleware holds the full application state. This allows the middleware to take memory snapshots and rollback to previous states. If dynamic allocation is required, it is necessary to hold a finite memory pool in `G` with a custom allocator. In our guiding example of Figure 3, 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

When an application generates events, they need to propagate in the network so that all peers behave the same. Note that application events may (or may not) differ from low-level local system events. The middleware predefines two events for all applications: `P2P_EVT_START` represents the beginning of the simulation, and `P2P_EVT_TICK` is generated every frame. The other application-specific events must be declared by the programmer in an enumeration. In our example, we define a new event `P2P_EVT_KEY` to represent key presses:

```
enum {
    P2P_EVT_KEY = P2P_EVT_NEXT // next to predefined
};
```

3.1.3 Initialization Callback

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

```
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` every frame or event occurrence, which should affect the simulation state `G` deterministically. An occurring event may be passed as argument to the callback, which 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: 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 separated from the rendering callback `cb_ren` [13], 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_ren (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 callback `cb_evt` is called by the middleware in real time, whenever a local SDL event occurs. This callback has two goals: (a) map and propagate application events, and (b) provide instant feedback to the user. Not all local low-level events need to correspond to application events in the network. The callback is allowed to perform side effects, such as modifying the network topology, or exhibiting instant feedback on the screen. In our example, we only generate application events for the 5 key events of interest, but we also exhibit the pressed key in real time as a visual feedback:

```
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 Middleware Orchestration

We now 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.

3.2.1 Event Dissemination

As illustrated in Figure 4, events propagate between peers with a timestamp scheduled to the future with an extra delta Δ such that all peers are able to apply them in sync. To prevent dissemination cycles, each peer keeps a vector with the highest timestamp it received from each other peer, such that lower numbers can be ignored when received. Since we rely on TCP connections, it is not possible to receive out-of-order timestamps attached to a given source peer. When the middleware receives a higher number from a given peer, it updates its vector position and triggers an event propagation to neighbours. As mentioned in Section 3.1, we assume that peers assign themselves unique identifiers in a contiguous range, such that they fit in a simple vector of timestamps.

When receiving an unseen event, the middleware also enqueues it in timestamp order and calls the application callback `cb_sim` at the appropriate time. Peers must keep the full event queue in memory for two reasons: (a) receiving a late event may reorder it and require a time travel, and (b) peers that join the network need to receive all events. Therefore, the middleware keeps a queue of "packets", which includes the event and additional metadata:

```
typedef struct {
    uint8_t  src;       // source peer
    uint32_t tick;      // tick to apply
    p2p_evt  evt;       // event id + payload
} p2p_pak;

struct {
    int i;  // next event to apply in real time
    int n;  // number of events in the queue
    p2p_pak buf[MAX];   // queue of packets
} PAKS;
```

Considering the peer-to-peer network as a whole, each event packet is replicated to all neighbours of all peers. This results in a theoretical limit of quadratic messages if all peers were connected to all peers. In contrast, as discussed in Section 2.1, a centralized solution only requires a packet to be sent to a server, which then broadcasts it to all clients.

3.2.2 The Main Loop

The middleware main loop `p2p_loop` is responsible for calling the application callbacks, and also controlling its timeline. The loop continuously checks for network packets, local inputs, and time ticks, as follows:

```
01  void p2p_loop (...,cb_ini,cb_sim,cb_ren,cb_evt) {
02    cb_ini(1);    // initialization
03    cb_sim(<P2P_EVT_START>);
04
05    while (<app-running>) {
06      if (<next-network-packet>) {
07        // fast forward if I'm late
08        if (<packet-sync-future>) {
09          p2p_travel(<now>, <pak-sync-time>, <vel>);
10        }
11
12        // travel back & forth if I'm early
13        if (<packet-evt-past>) {
14          p2p_travel(<now>, <pak-evt-time>, <vel>);
15          p2p_travel(<pak-evt-time>, <now>, <vel>);
16        }
17
18        // real time if I'm on time
19        if (<packet-evt-ok>) {
20          cb_sim(<packet-evt>);
21          cb_ren();
22        }
23      }
24
25      if (<next-local-input>) {
26        // propagate local event
27        p2p_evt evt;
28        if (cb_evt(&evt)) {
29          p2p_propagate(<future>, &evt);
30        }
31      }
32
33      // simulate tick in real time
34      if (<next-local-tick>) {
35        cb_sim(<P2P_EVT_TICK>);
36        <application-snapshot-every-second>
37        cb_ren();
38      }
39    }
40
41    cb_ini(0);    // finalization
42  }
```

The function receives the application callbacks as arguments (*ln. 1*) (Section 3.1). We first (and last) call `cb_ini` to initialize (and finalize) the application (*ln. 2,41*). Then, we call `cb_sim` (*ln. 3*), passing the event `START` to start the application in real time (Sections 3.1.2 and 3.1.4). The main loop (*ln. 5–39*) checks continuously for network packets (*ln. 6–23*), local input events (*ln. 25–31*), and time ticks (*ln. 33–38*).

Regarding network packets, if we receive a time synchronization packet `SYNC` from the future (*ln. 7–10*), then we fast forward the application to catch up in time (Section 3.b). If we receive an event that should have been applied in the past (*ln. 12–16*), then we travel back and forth (Section 3.a). Otherwise, we just apply the event in real time (*ln. 18–22*) (Section 3.1.4). We detail the function `p2p_travel` in Section 3.2.3. Regarding local inputs, we call `cb_evt` to signal if the event should be disseminated (*ln. 26–30*), also transforming it into an application event (Section 3.1.6). Regarding time ticks, we call `cb_sim` and `cb_ren` in real time (*ln. 33–38*). The middleware also takes periodic snapshots of the application state to optimize time travelling (Section 3.1.1).

3.2.3 The Time Machine

The last important mechanism of the middleware is the time machine, which allows to move the application back and forth in time:

```
01  void p2p_travel (int from, int to, int ms) {
02    for (i=from..to) {
03      int bef = <tick-of-snapshot-before-i>
04      <restore-snapshot-at-bef>
05      for (j=bef..i) {
06        cb_sim(<tick-or-evt-at-j>)
07      }
08      cb_ren();
09      <delay-ms>
10    }
11  }
```

The function `p2p_travel` time travels the application, starting at tick `from`, tick by tick (*ln. 2–10*), until it reaches tick `to`. The loop can travel in both directions, i.e., `from` may be higher than `to`. At each step, we need to find the tick `bef` with the closest past snapshot (*ln. 3*), restore it (*ln. 4*), and than simulate the remaining steps from the snapshot to the desired tick (*ln. 5–7*). We also exhibit each step with a small `ms` delay (*ln. 8–9*) so that users experience smooth time transitions.

The step delay is dynamically adjusted such that the whole time travel loop takes at most 1 second to complete. Note that during this period, the application is frozen, and new inputs are enqueued for further processing.

3.2.4 Middleware Summary

This section provided an overview of the middleware implementation to answer how it ensures the desired properties in concrete terms as follows: (a) peers meet event deadlines either travelling back and forth or in real time (*ln. 12–16* or (*ln. 18–22*) / 3.2.2); (b) peers synchronize their clocks with periodic `SYNC` packets that fast forward late peers (*ln. 7–10* / 3.2.2); and (c) peers can join the network at any time and still catch up in time by receiving the full event history (Section 3.2.1).

As discussed in Section 3.1, the middleware imposes some restrictions, which are discussed as follows:

- **Contiguous unique identifiers:** We assume that peers have unique identifiers in a contiguous range, which allows us to hold the peers database in a vector. Nevertheless, it is possible to generate UUIDs based local information only [15], and to use a hash table as database instead.

- **Deterministic API:** Since simulations must be equally reproduced in all peers, non-deterministic APIs are forbidden. This is a fundamental limitation that cannot be easily overcame. Note that the alternatives discussed in Section 2 either impose the same limitation [2, 1], or can be even more restrictive allowing only CRDTs [5].

- **Non-malicious peers:** As detailed in Section 3.1.6, note that the API to disseminate an event in the network is as simple as a variable assignment. Therefore, a malicious peer could SPAM or inject erroneous events to break the application. Again, this is a fundamental limitation that is also shared with the alternatives discussed in Section 2.

- **Static memory footprint:** The middleware takes periodic snapshots of the whole application state in order to optimize time machine simulations and resynchronize peers. This mechanism imposes two challenges: first, the snapshot must be small and determined, which implies a contiguous static region; second, the history of snapshots must be stored in all peers. Currently, we allocate periodic snapshots assuming that the memory does not deplete until the application terminates. Alternative policies could favor recent snapshots or simply impose a limit that reject clients that are too late. Regarding the static limitation, we mentioned the possibility of a custom allocator, which is illustrated as follows:

```
struct {
    // FIXED REGION
    int x,  y;   // position
    int dx, dy;  // direction speed
    // VARIABLE REGION
    char heap[HEAP_SIZE];
} G;

T* obj = my_malloc(G.heap, <size>);
<...>
my_free(obj);
```

The variable region is still technically static, but is dynamically reshaped internally with the custom allocator `my_alloc` & `my_free`.

4. Evaluation

In this section, we evaluate if our design meets the goals to ensure 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. We perform simulations varying the network latency and peer churn, as well as the frequency and deadlines of events. We then measure (a) the recurrence of time travels, (b) the real-time pace of peers, and (c) the recoverability from churns, such that they validate our goals above, respectively.

Regarding (a), if a peer experiences a time travel, it means that it received an event after its deadline. We then count all time travel occurrences in our evaluation. Regarding (b), we use an absolute simulation clock to track, in real time, the execution pace of each peer. We then count the mismatches between the clocks in our evaluation. Regarding (c), we make peers leave and join the network periodically, possibly creating partitions. We then assert that they all catch up in real time within a short period in our evaluation.

We perform simulations with 21 peers using the mixed topology of Figure 5, with peers organized in straight lines and cycles. The topology has an average of 5 hops between peers (e.g., `4-16` are 7 hops away). We run the application main loop at 50 FPS (20ms per frame) and calculate the middleware overhead at each iteration, which is 9us in the average, corresponding to less than $1/2000$ of the CPU time. In order to have an absolute clock, we use a single machine to simulate all peers, allowing us to merge their logs into a single file with a comparable shared timeline. We use *NetEm* [16] to simulate the network latency with a normal distribution. We vary (i) the network latency ($5-500ms$ between each peer), (ii) the rate of events ($5-200$ evt/sec), and (iii) the delay multiplier ($1-5x$). Figure 6 shows the event delays considering the network latency and chosen multiplier. For instance, a 50ms latency with a 2x multiplier results in 500ms event delays, i.e., each event is triggered with this delay to compensate the given latency and fixed average of 5 hops. These variations result in 108 combinations. We executed each combination 3 times for 5 minutes each. We also make some modifications to evaluate peer churn, which requires to re-execute the simulation. The experiments take around 40 hours to complete.

We now detail the results and how we instrument the middleware implementation to measure the properties above:

(a) Recurrence of time travels: Every time a peer is ahead in time and needs to travel back, we output its id and how much ticks it needs to travel back. At the end, we measure the percentage of the sum of all time travel ticks over the total simulation ticks. As an example, if all peers sum 1000 time travel ticks over 100k accumulated simulation ticks, then the final measure is 1%. Figure 7 presents the results considering all combinations of parameters (i), (ii), and (iii) above. The colors indicate the feasibility of each tested scenario (*white=good*, *yellow=moderate*, *red=poor*, *black=faulty*). For instance, the three framed rectangles (0.1, 3.1, and 7.2) in the figure represent the scenarios with 50ms network delay, generating 25 evt/min. Each rectangle uses a different delay multiplier, showing a good performance for 5x, moderate for 2x, and poor for 1x.

(b) Real-time pace of peers: All peers output their current ticks continuously, allowing us to compare them in the shared timeline. Every time we see a greater tick never seen in the timeline before, we count smaller ticks from other peers, which constitute time violations. As an example, if all peers sum 1000 of such smaller ticks over 100k accumulated simulation ticks, then the final measure is 1%. We also start each peer with a random delay of up to 10s to force a synchronization mismatch at the beginning. Considering all scenarios, the average is negligibly under 0.2%, with no significant variance. Therefore, we omit the resulting table akin to Figure 7), which would not provide further insights.

(c) Correctness of unstable peers: We tweak our simulation to make each peer remain 40s online and 20s offline in the average, including peers `9-11` in the middle of the network. This creates partitions in the network, which generate events out of sync that require constant resynchronization between peers. Then, to evaluate the network correctness under such high churn, we count how much time a recovering peer takes to catch up in real time: When a peer becomes online, we take the current maximum tick considering all reachable peers, and then count the time the recovering peer takes to catch up. We also assert that they all reach the same final state eventually. Considering all scenarios, the average is 1.3 seconds of recovering time, with no significant variance. Therefore, we also omit the resulting table (akin to Figure 7).

Our results show that goals (b) and (c) are met in all scenarios consistently, and we consider them nearly optimal: Regardless of the network latency, event rate, and delay multiplier, the variation in the pace of peers is under 0.2% (goal b), while recovering peers take 1.3 seconds to catch up in time (goal c). Note that the middleware makes smooth time transitions in small steps that take 1 second to complete, resulting in no more than 2 time travels to recover peers.

Considering goal (a), we need to take a closer look at Figure 7. The color patterns indicate clear boundaries for the scenarios that can meet event deadlines with the desired performance. For instance, high event delays (column 5x) make all scenarios viable, even with high latency and event frequency. As an application example, chats are not affected by high event delays. Nevertheless, the higher are the event delays, the less applications suit the middleware. For the lowest possible delay (column 1x), we see that the middleware can either handle low latency with high event frequency (e.g., 5ms with 100evt/min) or high latency with low event frequency (e.g., 100ms with 5evt/min). The color patterns in the table provide the necessary information to confront an application with its feasibility or desired performance.

Considering the metrics we evaluate, we now compare our work with the existing solutions of Section 2.1:

(a) Recurrence of time travels: In Croquet, the centralized reflector issues periodic ticks, which represents the only authority over time. On the one hand, clients never need to travel back in time, but on the other hand, the reflector represents a single point of failure. GALS imposes the same tradeoffs, relying on a central server to provide a single authority over time. CRDT operations can be applied in any order, which also excludes time travels entirely. However, the notion of a shared real-time timeline cannot exist, which

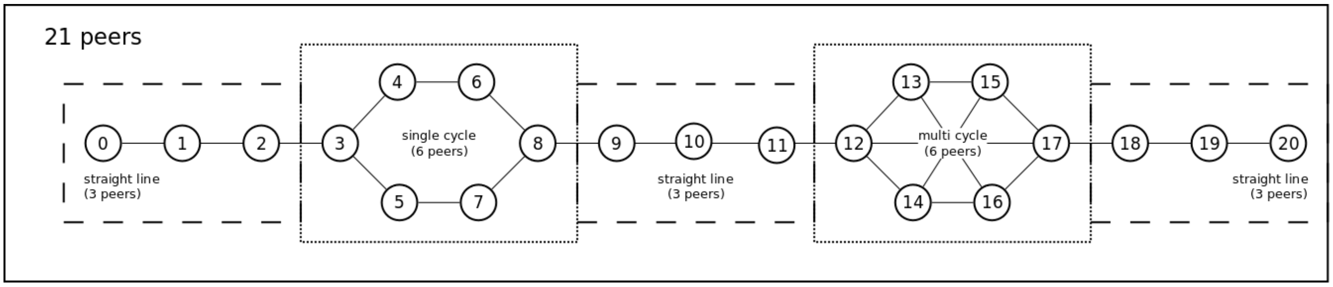


Figure 5. Network topology: 21 peers organized as a straight line $(0-3)$, a simple cycle $(3-8)$, a second straight line $(8-11)$, a full connected cycle $(12-17)$, and a third straight line $(17-20)$.

Multiplier → Latency ↓	1x	2x	5x
5ms	25	50	125
25ms	125	250	625
50ms	250	500	1250
100ms	500	1000	2500
250ms	1250	2500	6250
500ms	2500	5000	12500

Figure 6. Event delays in *ms* considering the network latency, delay multiplier, and the average of 5 hops between peers from Figure 5.

precludes bit-identical behavior across peers.

(b) Real-time pace of peers: Croquet stipulates a global tick period which all clients must comply with. However, clients with poor connections experience constant freezes and cannot advance in real time. As a concrete example, a peer that experiences a latency of 500*ms*, while the reflector issues ticks every 100*ms*, will freeze for 400*ms*, even if no events occur in the network. GALS supports independent timelines for clients, but at the cost of applying events with a greater delay in comparison to Croquet. With CRDTs, although peers advance independently, the notion of real time does not exist.

(c) Correctness of unstable peers: Since clients are connected directly with the server, Croquet and GALS have no issues with unstable clients. When reconnecting, a client just needs to synchronize pending events from the server and apply them in order. However, as discussed before, the server represents a single point of failure. CRDTs are also immune to unstable peers, since peers just need to synchronize events and apply them in any order, but again missing the notion of real-time behavior.

5. Conclusion

We propose a middleware to program *symmetric peer-to-peer applications* in which interacting decentralized instances conform to identical behavior. The middleware API shares similar limitations with centralized alternatives [1, 2], being restricted to deterministic and stateless calls, and only supports pre-allocated memory. Our main contribution is a "time machine" based on memory snapshots and deterministic simulation, allowing to decentralize the network timeline such that conflicting peers can rollback and resynchronize their state. We assume that peers are non-malicious and that they can assign themselves unique identifiers in a contiguous range.

The middleware can handle applications with over 20 peers in a heterogeneous topology under high churn. We show that peers can (a) disseminate events and meet deadlines globally, (b) advance in sync in real time, and (c) leave and join the network and remain symmetric. In our experiments, we vary the network latency and the rate and deadline of events. We than measure the recurrence of time travels in the network as a representative of quality. As a result, we present a feasibility table to indicate the application scenarios that the middleware can handle. For instance, in a simulation scenario of 25 events per minute with a network latency of 50*ms*, the peers need to roll back and resynchronize only 3% of the time.

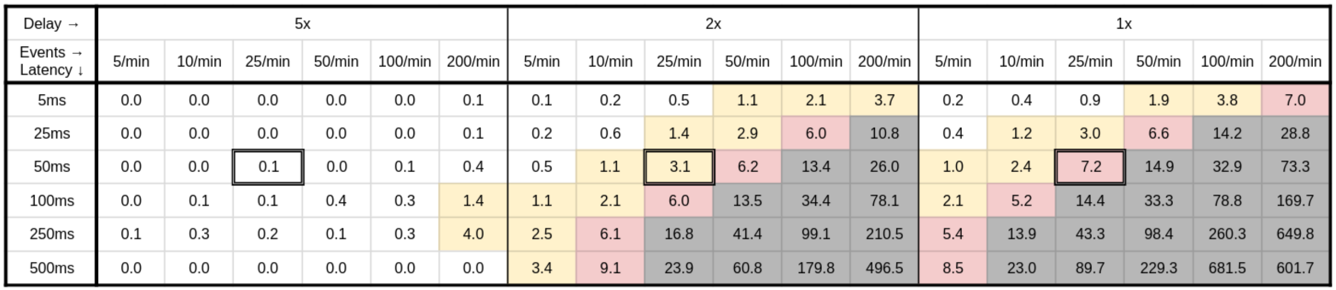
References

[1] SANT'ANNA, F.; SANTOS, R.; RODRIGUEZ, N. Symmetric distributed applications. In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. [S.l.: s.n.], 2021. p. 41–50.

[2] SMITH, D. A. et al. Croquet-a collaboration system architecture. In: IEEE. *First Conference on Creating, Connecting and Collaborating Through Computing, 2003. C5 2003. Proceedings.* [S.l.], 2003. p. 2–9.

[3] THEOTOKIS, S. A.; SPINELLIS, D. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, Association for Computing Machinery, New York, NY, USA, dez. 2004. ISSN 0360-0300.

[4] OHSHIMA, Y. et al. An experiment in live collaborative programming on the croquet shared experience platform. In: *Companion Proceedings of the 6th International Conference on the Art, Science, and Engineering of Programming*. [S.l.: s.n.], 2022. p. 46–53.



Delay →	5x						2x						1x					
Events → Latency ↓	5/min	10/min	25/min	50/min	100/min	200/min	5/min	10/min	25/min	50/min	100/min	200/min	5/min	10/min	25/min	50/min	100/min	200/min
5ms	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.2	0.5	1.1	2.1	3.7	0.2	0.4	0.9	1.9	3.8	7.0
25ms	0.0	0.0	0.0	0.0	0.0	0.1	0.2	0.6	1.4	2.9	6.0	10.8	0.4	1.2	3.0	6.6	14.2	28.8
50ms	0.0	0.0	0.1	0.0	0.1	0.4	0.5	1.1	3.1	6.2	13.4	26.0	1.0	2.4	7.2	14.9	32.9	73.3
100ms	0.0	0.1	0.1	0.4	0.3	1.4	1.1	2.1	6.0	13.5	34.4	78.1	2.1	5.2	14.4	33.3	78.8	169.7
250ms	0.1	0.3	0.2	0.1	0.3	4.0	2.5	6.1	16.8	41.4	99.1	210.5	5.4	13.9	43.3	98.4	260.3	649.8
500ms	0.0	0.0	0.0	0.0	0.0	0.0	3.4	9.1	23.9	60.8	179.8	496.5	8.5	23.0	89.7	229.3	681.5	601.7

Figure 7. The $5x$, $2x$, and x delay regions each contains 36 measures for multiple configurations of event rate ($5-200$ *evt*/*min* and network latency ($5-500ms$). The color thresholds (*white*, *yellow*, *red*, *black*) are arbitrary but help to distinguish between regions of interest. The experiments uses 1 process for each peer in a single Linux machine (*i7/16GB*).

[5] SHAPIRO, M. et al. Conflict-free replicated data types. In: SPRINGER. *Symposium on Self-Stabilizing Systems*. [S.l.], 2011. p. 386–400.

[6] KLEPPMANN, M. et al. Local-first software: you own your data, in spite of the cloud. In: *Proceedings of Onward'19*. [S.l.: s.n.], 2019. p. 154–178.

[7] KLEPPMANN, M.; BERESFORD, A. R. A conflict-free replicated json datatype. *IEEE Transactions on Parallel and Distributed Systems*, IEEE, v. 28, n. 10, p. 2733–2746, 2017.

[8] HARDENBERG, P. van; KLEPPMANN, M. Pushpin: Towards production-quality peer-to-peer collaboration. In: *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*. [S.l.: s.n.], 2020. p. 1–10.

[9] LAMPORT, L. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE transactions on computers*, IEEE Computer Society, v. 28, n. 09, p. 690–691, 1979.

[10] HOWARD, H.; MORTIER, R. Paxos vs raft: Have we reached consensus on distributed consensus? In: *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*. [S.l.: s.n.], 2020. p. 1–9.

[11] GOMES, V. et al. Verifying strong eventual consistency in distributed systems. In: . [S.l.]: ACM New York, NY, USA, 2017. v. 1, p. 1–28.

[12] ENGBLOM, J. A review of reverse debugging. In: IEEE. *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*. [S.l.], 2012. p. 1–6.

[13] SCHUSTER, C.; FLANAGAN, C. Live programming for event-based languages. In: *Proceedings of the 2015 Reactive and Event-based Languages and Systems Workshop, REBLS*. [S.l.: s.n.], 2015. v. 15.

[14] BURCKHARDT, S. et al. It's alive! continuous feedback in ui programming. In: *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. [S.l.: s.n.], 2013. p. 95–104.

[15] ABERER, K.; DATTA, A.; HAUSWIRTH, M. Efficient, self-contained handling of identity in peer-to-peer systems. *IEEE Transactions on Knowledge and Data Engineering*, IEEE, v. 16, n. 7, p. 858–869, 2004.

[16] HEMMINGER, S. et al. Network emulation with netem. In: CITESEER. *Linux conf au*. [S.l.], 2005. v. 5, p. 2005.