A Middleware for Symmetric Peer-to-Peer Applications

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

Abstract—In real-time networked applications, such as collaborative documents, users can interact remotely and yet share the exact same experience as if they were using 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. Peers are allowed to join and leave the network at any time. Application developers must adhere to a deterministic and stateless API. 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 states. We show that the middleware can handle applications with over 20 peers in heterogeneous topologies under high churns. For instance, in a scenario with a network latency of 50ms and 25 events per minute, the peers need to roll back only 3% of the time to remain fully symmetric.

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

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 collaborative 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. This problem relates to consistency in distributed systems, but with the additional requirement of a time deadline to reach consistency [2].

A common approach towards symmetric applications is to introduce a middleware to orchestrate events and time in the network [1], [3]. 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. Therefore, applications may experience failures, unless users are permanently connected, with no partitions in the network, particularly from the server. In contrast, a decentralized solution could be resilient to partitions, and ultimately, even work offline as promoted by local-first software [4].

In this work, we propose a middleware for *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 [5]. Events are flooded in the graph and are applied with a small deadline delay to compensate the network latency. To deal with events received out of order or too late, a time machine can rollback a peer to a previous state and reapply the events in order, such that a single consistent timeline is established across the network. 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. Our middleware ensures *perceptive consistency* [2], in which an event triggered by any peer at time t is perceived by all peers simultaneously at time $t + \Delta$. As the main limitation, the middleware cannot ensure *instantaneity* for events in most scenarios, which would require Δ to be less than a user-observable threshold [2]. We also assume that peers are non malicious in the sense that they do not lie about the time in which events occurred locally.

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 goals (b) and (c) 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 goal (a), we analyze the scenarios in which the middleware can meet event deadlines with the desired performance. As an example, peers need to roll back only 3% of the time under a network latency of 50ms and 25 events per minute. We consider this overhead negligible for soft real-time applications that do not depend on instantaneous network feedback, such as chats, watch parties, and turn-based games.

In Section 2, we revisit existing solutions for symmetric distributed applications. 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.

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

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

2 Related Work

In this section, we revisit existing solutions for symmetric distributed applications, which are not necessarily peer to peer. 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 three selected works regarding these aspects against our work. One of these approaches relies on CRDTs [6], and is discussed in separate. At the end of this section, we also discuss related work on local and distributed time machines.

2.1 Symmetric Distributed Applications

Croquet [7], [3], [8] guarantees bit-identical real-time behavior for users in collaborative distributed environments. As detailed in Figure 2, a centralized *reflector* issues fixed periodic ticks (e.g., every 50ms), 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. This mechanism is similar to *bucket synchronization* [9], in which events are assigned to time buckets to be applied later. 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 request 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 broadcasts and outcomes depend on the central server to be online (c).

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*), resulting in dynamic deadlines according to the slowest client. On the one hand, ticks do not generate any traffic, and clients experience 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 broadcasts and outcomes still depend on the central server to be online (c).

In both solutions, the network advances together in real time as a whole, 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 exact same state, and observe bit-identical steps.

In this work, as indicated in Figure 1, our goal is to support (a) peer-to-peer real-time symmetric execution, while still (b) tolerating out-of-order, and (c) offline event generation. As detailed in Figure 2, peers have independent timelines and can communicate events directly to each other. Event deadlines 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. This technique is also known as *log-based rollback-recovery* [10], but applied in the context of real-time collaborative applications. Like Croquet and GALS, peers compute events in sequence using deterministic and stateless calls only.

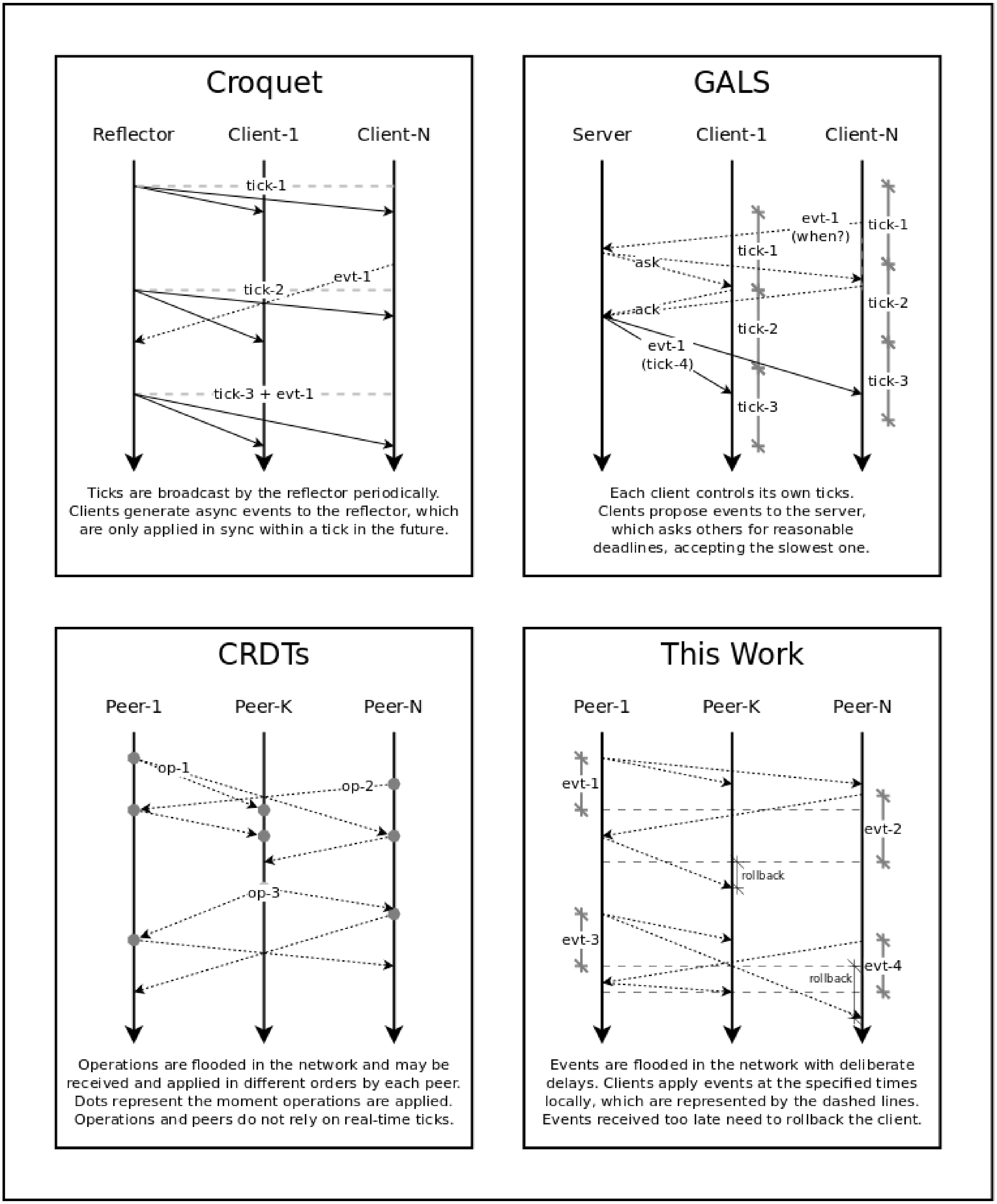
As it is the case for the discussed work in this section, we consider that middlewares for symmetric applications must comply with *perceptive consistency* [2], as follows:

"**Perceptive consistency** [11], [12]. If any operation is executed on process i at time t, it must be executed on process j at t as well. At a high-level point of view, it indicates that for a group of users interacting with each other, if a user takes an action, then the action will be perceived by everyone simultaneously (though the action might be perceived at a time later than the time at which it was taken by the user). It is also referred to as absolute consistency [11]."

2.2 Conflict-free Replicated Data Types

An antagonistic approach to deterministic reactions to events is to model the applications with Conflict-free Replicated Data Types [6]. 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 originating peers are offline. Note that it is not possible to timestamp commutative operations under 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 [13]. On the one hand, the main advantage of CRDTs is to support 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 [14]. As indicated in Figure 1, CRDTs work on (a) decentralized networks, in which (b) peers advance independently, and in which (c) operations can be applied immediately, even while offline. Automerge [15] along with its accompanying Hypermerge peer-to-peer infrastructure [16] is an example of decentralized CRDTs.

The main advantage of CRDTs is that they can be designed in such a way that concurrent modifications to shared state can preserve intent [17], regardless of the order in which they are applied. This is not the case for the symmetric application middlewares described in the previous



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. Clents 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.

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

section. Nevertheless, programmers could still use CRDTs on top of a middleware to preserve intent, since they are nothing more than data structures and merging algorithms.

Contrasting with perceptive consistency, CRDTs provide *delayed consistency* [2], as follows:

"**Delayed consistency** [11]. This is a relaxation of perceptive consistency. In this model, for any operation issued by host i, say at time t, it takes effect instantaneously on host i. On a remote host j, it is executed at time $t + (i, j)$, where (i, j) is the phase different between processes i and j, and it is a parameter specific to these two processes. Unlike perceptive consistency, this model does not force all processes to execute the same operation at the same time. Rather, it makes sure that the state of objects operated by one process be viewed at another process with a consistent delay."

2.3 Time Machines

Regarding decentralized time machines, the *Time Warp* mechanism [18] rollbacks the memory state of a replica when it receives an event from the past, and also broadcasts "anti-messages" for all events that were incorrectly applied. Our work assumes that all events are correct, and reapplies them relying on deterministic simulation.

Fusion [19] 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.

Regarding local single-user time machines, the video game Braid[1] 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 [20] 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 mutation from rendering [21].

3 The Middleware & Programming API

Our middleware is written in C and relies on the library *SDL*[2] to deal with the network, timers, and input events, such as mouse clicks and key presses. The full source code[3] is around 500 lines of code.

We guide our discussion through the didactic example of Figure 3, in which multiple users control the direction of a moving rectangle using the arrow keys. The six instances are connected in a peer-to-peer network, such that most peers can only communicate indirectly to each other. The application is symmetric in the sense that if any user presses a key, a corresponding event is broadcast and applied in all peers 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 "bit-identically" at the very same position. This is only possible because the middleware ensures that all peers shares the same timeline of events, in which only deterministic operations are applied.

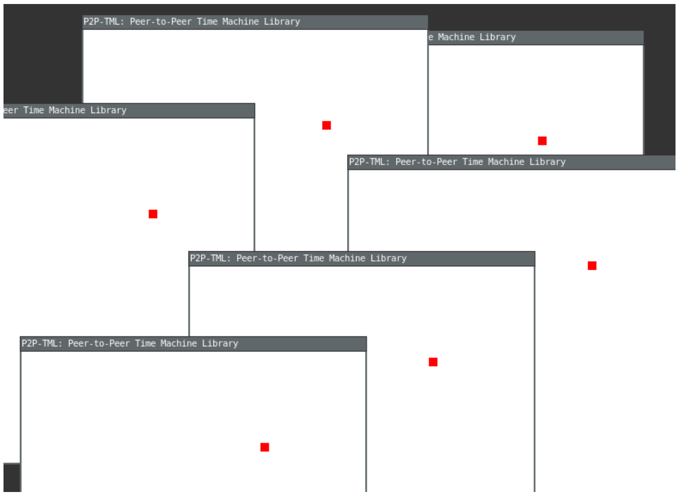

P2P-TML: Peer-to-Peer Time Machine Library
e Machine Library
eer Time Machine Library
P2P-TML: Peer-to-Peer Time Machine Library
P2P-TML: Peer-to-Peer Time Machine Library
P2P-TML: Peer-to-Peer Time Machine Library

Fig. 3. **An application example:** 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

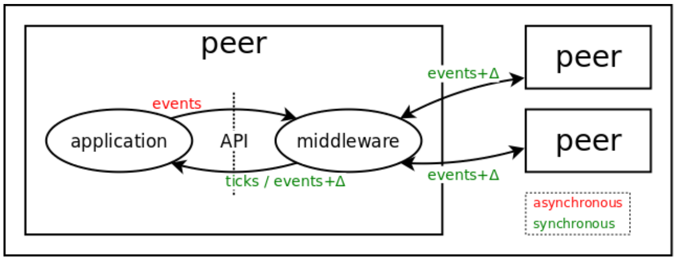


Fig. 4. **The middleware architecture:** Applications generate asynchronous events and communicate through an API with the middleware. The middleware orchestrates the peers, synchronizing their ticks locally and broadcasting events with a deadline delay (Δ).

In Figure 4, a peer is depicted as an application using 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 deadline delay (Δ) scheduled to the future (aka *local lag* [22]]), such that all peers can satisfy. The middleware controls the execution of the application by issuing ticks and synchronized events. Note that ticks need not to be broadcast, only Δ events.

The main job of the middleware is to deal with the uncertainties of the network in order to provide a consistent timeline and 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

[1] Braid: http://braid-game.com/
[2] SDL: http://www.libsdl.org
[3] Code: http://github.com/fsantanna/p2p-tml/

in real time, and (c) can leave and join the network and remain symmetric:

a) **Event Deadlines:** Due to the inherent latency of networks, a peer (through the middleware) generates events with a deliberate 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, which characterizes an optimistic approach to choose Δ, since it assumes that inconsistencies may occur [2]. 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 the whole application execution and event occurrences from the beginning. As an optimization, the middleware takes periodic snapshots locally to avoid full simulation.

b) **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 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. To avoid abrupt visual changes, the middleware makes smooth time transitions that take up to $1s$ to complete.

c) **Peer Churn:** Considering that we target dynamic peer-to-peer networks, it is important that the middleware can handle arrival and departure of peers seamlessly. Note that high churns 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)).

As a limitation of our work, regarding items (a) and (b), we assume that peers never lie about their local clocks, which would affect event timestamps, allowing a peer to modify the global timeline with nonsensical information. Therefore, we assume a permissioned peer-to-peer network with an external mechanism for authentication.

3.1 The Programming API

The middleware expects to take full control of the application 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 (e.g., a combination of its MAC & IP addresses). The FPS must be the same in all peers to ensure bit-identical simulation.

Note that using callbacks is already a common technique for event-driven programming, such as in Games, GUIs, and interactive applications in general, sharing similarities with *MVC*, *Publish-Subscribe*, and *Observer* patterns [23], [24]. Furthermore, the default run-to-completion semantics of callbacks is key to ensure the desired determinism, since unlike threads, they run sequentially and atomically from start to end [25], [26].

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. In our guiding example of Figure 3, the application only needs to hold the rectangle position and direction:

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

If dynamic allocation is required, it is necessary to hold a finite memory pool in `G` with a custom allocator. This is a limitation of C, since languages with serialization and metaprogramming offer better mechanisms to take memory snapshots. Another alternative is to provide a serialization callback, which we left out for future work.

3.1.2 Application Events

When an application generates events, they need to be broadcast to 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 three events for all applications: `P2P_EVT_SYNC` represents synchronization events (Section 3.b). `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 // KEY after predefs
};
```

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 of 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. Occurring events are passed as arguments to the callback, which must never perform any side effects (outside `G`), such as stateful calls or rendering frames. In our example, we modify the state of the rectangle as follows: on `START`, reset its position and speed; on `TICK`, increment its position based on the current speed; and on `KEY`, modify its speed:

```
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;
                <...> // similar 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.

The simulation callback is the most sensitive API of the proposed middleware, since it is responsible for preserving the consistency of the global application state `G`, even under multiple executions from scratch. Since the code is written in C, which is unaware of the semantics of symmetric applications, note that it is the responsibility of the programmer to ensure that the state is completely reproducible. As discussed in Section 2, the typical approach, which we chose, is to rely on deterministic and stateless calls only.

3.1.5 Rendering Callback

The restriction for stateless calls is the reason why the rendering callback `cb_ren` must be separated from `cb_sim` [20], which would otherwise render the screen multiple times while time travelling. Instead, the middleware calls `cb_ren` to render the screen after each frame event, but only in real time. 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 invoked by the middleware only in real time, whenever a local SDL event occurs. This callback has two goals: (a) to map and broadcast application events, and (b) to provide instant feedback to the user. Not all local low-level events need to be broadcast in the network, only those mapped as application events, as described in Section 3.1.2. 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 keys 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 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.

3.2.1 Event Broadcasting

As illustrated in Figure 4, events are broadcast between peers with a deadline Δ scheduled to the future, such that all peers are able to apply them in sync. To prevent broadcast cycles, each peer keeps a set of the highest timestamps 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 this number and triggers a broadcast to neighbours.

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;
```

3.2.2 The Main Loop

The middleware main loop `p2p_loop` is responsible for invoking 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        // broadcast local event
27        p2p_evt evt;
28        if (cb_evt(&evt)) {
29          p2p_bcast(<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*). 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. 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. If we receive an event that should have been applied in the past (*ln. 12–16*), then we travel back and forth. Otherwise, we just apply the event in real time (*ln. 18–22*). 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 broadcast (*ln. 26–30*), and transformed into an application event. 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.

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.

3.2.4 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 in real time or travelling back and forth (Section 3.2.2: *ln. 18–22* or *ln. 12–16*); (b) peers synchronize their clocks with periodic `SYNC` packets which fast forward late peers (Section 3.2.2: *ln. 7–10*); 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).

Two facts ensure consistency in the network: (a) peers share a global timeline of events, and (b) they go through bit-identical steps However, since the clocks may diverge temporarily, we must rely on simulation rollbacks. Therefore, the key to preserve consistency in the presence of rollbacks is to restrict our programming API to deterministic and stateless calls only. This way, it becomes impossible to diverge from other peers by inadvertently re-executing stateful file operations multiple times (or at different times).

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 opted to mix three abstract arrangements: (i) straight lines, and (ii) weakly and (iii) strongly connected cycles. Even though this topology is arbitrary, we can expect that concrete scenarios with subsets of these arrangements will have at least the same performance of our experiments. For instance, a straight line of 15 peers is a subset of our topology, as well as two cycles separated by a single peer.

We run the application main loop at 50 FPS (20ms per frame) and calculate the middleware overhead at each iteration. We found a negligible overhead of $9us$ in the average, which corresponds to less than $1/2000$ of the CPU time. This means that an application using the middleware would perform all synchronization upkeep and still have almost 100% of the CPU time available for its actual domain logic. 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* [27] 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 deadline Δ multiplier ($1-5x$). Figure 6 shows the Δs considering the network latency and chosen multiplier. For instance, a $50ms$ latency with a $2x$ multiplier results in $\Delta = 500ms$, i.e., each event is triggered with this deadline delay to compensate the given latency and fixed average of 5 hops ($5 \times 50 \times 2 = 500$).

All simulation 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 of interest 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 latency, generating 25 evt/min. Each rectangle uses a different Δ multiplier, showing a good performance for $5x$, moderate for $2x$, and poor for $1x$.

(b) Real-time pace of peers: All peers output their ticks continuously, which we compare in the shared timeline. Every time we see a greater tick never seen in the timeline before, we start to 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, as follows: 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, which are 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). As discussed in Section 3.b, the middleware may take up to 1 second to catch up, resulting in no more than 2 time travels to recover a peer.

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 deadlines (column $5x$) make all scenarios viable, even with high latencies and event frequencies. As an application example, chats are not

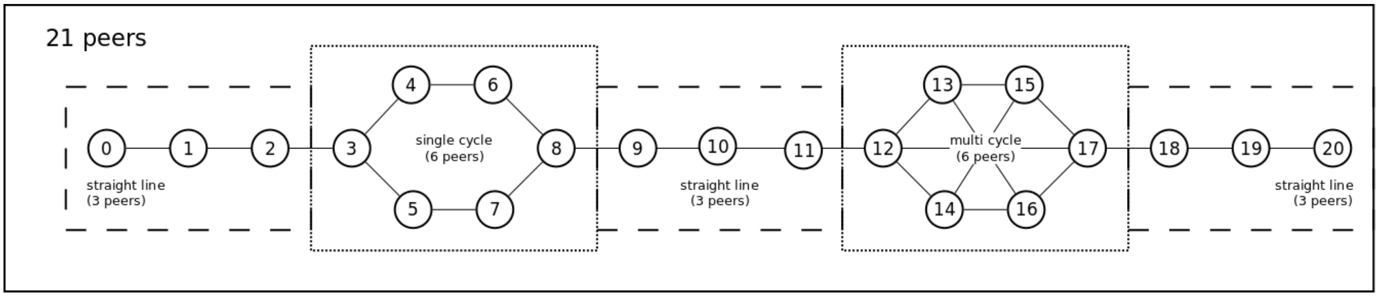


Fig. 5. Network topology: 21 peers organized as a straight line ($0-3$), a simple cycle ($3-8$), a second straight line ($8-11$), a full connected cycle ($12-17$), and a third straight line ($17-20$).

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

Fig. 6. Event Δs in ms considering the network latency, deadline multiplier, and the average of 5 hops between peers from Figure 5.

affected by higher event deadlines. Nevertheless, the lower is the deadline, the less applications suit the middleware. For the lowest possible Δ (column $1x$), we see that the middleware can either handle low latencies with high event frequencies (e.g., $5ms$ with $100evt/min$) or high latencies with low event frequencies (e.g., $100ms$ with $5evt/min$). Note that the middleware cannot ensure *instantaneity* for events in most scenarios, which would require a very low Δ [2]. The middleware does not perform well under higher network latencies (e.g., over 250ms). This is expected since peers that are multiple hops away take too long to synchronize past events and may generate future events in the meantime, which results in time travels. In summary, the color patterns in the table provide the necessary information to confront an application with its feasibility or desired performance.

4.1 Discussion

Many aspects may affect the performance of symmetric distributed applications, such as the network latency, the number of nodes in the network, the rate of user events, and the application frame rate.

It is expected that centralized solutions exhibit a better performance in comparison to peer-to-peer applications, since nodes are only 2 hops away from each other. However, the main benefit of our work is that applications remain functional even under network partitions, which includes failures in (would be) servers. Taking this possibility to an extreme, a peer can still work normally even while completely offline for a period of time: Even though rollbacks are inevitable when reconnecting, their effects may be unnoticeable to final users if there is low activity or dependency between events.

As pointed out in column (c) of Figure 1, centralized solutions require that clients are permanently connected with the server, which is responsible to control the shared timeline of events.

The performance of Croquet [3] depends exclusively on the network latency, which restricts the application to a maximum FPS. For instance, since all frame ticks need to be transmitted over the network, a latency of 50ms supports at most 20 FPS. Also, the higher is the FPS, the higher is the network traffic and server load, which may also degrade performance. GALS [1] detaches latency from FPS, since each node has its own timeline. However, the protocol requires two round trips to the server, which degrades performance considerably. The authors claim to achieve a responsiveness of 350ms in applications with 25 nodes at 25 FPS, where 350ms represents the time it takes for all nodes to apply an event since its initial occurrence.

The performance of our middleware is measured in terms of rollbacks, i.e., the more the peers need to resynchronize and travel back in time, the lower is the performance of the application. In a similar scenario with 21 peers at 50 FPS, and average of 5 hops with 50ms latency, we show 3.1% of rollbacks with responsiveness of 500ms, which we classify as moderate performance (Figure 7, framed rectangle in the center). As expected, the peer-to-peer performance is lower in comparison to centralized solutions, but still viable in many scenarios.

Regarding CRDTs, recall that they only provide eventual consistency with no notion of a shared timeline in which peers go through identical steps. Therefore, measuring the performance of event responsiveness, maximum FPS, or clock synchronization is meaningless, given that events can be applied locally and broadcast using a best-effort policy.

5 Conclusion

We propose a middleware to program *symmetric peer-to-peer applications* in which interacting peers conform to identical behavior without a centralized coordination server. The middleware API shares similar limitations with centralized alternatives [1], [3], being restricted to deterministic and stateless calls. Our main contribution is a time machine that supports rollbacks based on memory snapshots and deterministic simulation. This allows us to decentralize the network timeline, since conflicting peers can resynchronize

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

Fig. 7. The squares show the percentage of time travels. The $5x$, $2x$, and x Δ regions each contains 36 measures for multiple configurations of network latencies ($5-500ms$) and event rates ($5-200\ evt/min$). 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 (*Ubuntu 21.04*, *i7/16GB*).

their state through time travels. The middleware provides *perceptive consistency* [2], which adds a deadline to reach consistency, ensuring that an event triggered by any peer at time t is perceived by all peers simultaneously at time $t + \Delta$.

The middleware can handle applications with over 20 peers in heterogeneous topologies and under high churns. We show that peers can (a) broadcast 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 quality proxy. As a result, we present a performance table to indicate the feasibility of diverse application scenarios. For instance, in a simulation scenario with a network latency of 50ms and 25 events per minute, the peers need to rollback only 3% of the time.

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] H. Liu, M. Bowman, and F. Chang, "Survey of state melding in virtual worlds," *ACM Computing Surveys (CSUR)*, vol. 44, no. 4, pp. 1–25, 2012.

[3] Y. Ohshima, A. Lunzer, J. Evans, V. Freudenberg, B. Upton, and D. A. Smith, "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*, 2022, pp. 46–53.

[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] S. Androutsellis-Theotokis and D. Spinellis, "A survey of peer-to-peer content distribution technologies," *ACM computing surveys (CSUR)*, vol. 36, no. 4, pp. 335–371, 2004.

[6] 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.

[7] 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.

[8] "Fusion: a high performance state synchronization networking library for unity," https://croquet.io/docs/croquet/, accessed: 2022-10-29.

[9] L. Gautier and C. Diot, "Design and evaluation of mimaze a multi-player game on the internet," in *Proceedings. IEEE International Conference on Multimedia Computing and Systems (Cat. No. 98TB100241)*. IEEE, 1998, pp. 233–236.

[10] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys (CSUR)*, vol. 34, no. 3, pp. 375–408, 2002.

[11] X. Qin, "Delayed consistency model for distributed interactive systems with real-time continuous media," *Journal of Software*, vol. 13, no. 6, pp. 1029–1039, 2002.

[12] N. Bouillot and E. Gressier-Soudan, "Consistency models for distributed interactive multimedia applications," *ACM SIGOPS Operating Systems Review*, vol. 38, no. 4, pp. 20–32, 2004.

[13] M. Letia, N. Preguiça, and M. Shapiro, "Crdts: Consistency without concurrency control," *arXiv preprint arXiv:0907.0929*, 2009.

[14] H. Sanjuán, S. Poyhtari, P. Teixeira, and I. Psaras, "Merkle-crdts: Merkle-dags meet crdts," *arXiv preprint arXiv:2004.00107*, 2020.

[15] 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.

[16] 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.

[17] G. Litt, S. Lim, M. Kleppmann, and P. van Hardenberg, "Peritext: A crdt for collaborative rich text editing," *Proceedings of the ACM on Human-Computer Interaction (PACMHCI)*, 2022.

[18] D. R. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 3, pp. 404–425, 1985.

[19] "Fusion: a high performance state synchronization networking library for unity," https://doc.photonengine.com/en-us/fusion, accessed: 2022-10-29.

[20] 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.

[21] 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.

[22] M. Mauve, "Consistency in replicated continuous interactive media," in *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, 2000, pp. 181–190.

[23] B. Meyer, "The power of abstraction, reuse, and simplicity: An object-oriented library for event-driven design," pp. 236–271, 2004.

[24] R. Nystrom, "Game programming patterns," 2014.

[25] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazieres, and R. Morris, "Event-driven programming for robust software," in *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, 2002, pp. 186–189.

[26] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[27] S. Hemminger *et al.*, "Network emulation with netem," in *Linux conf au*, vol. 5. Citeseer, 2005, p. 2005.