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 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 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 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.

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 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. 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 of order or too late, a distributed time machine can rollback peers 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 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.

	(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 propagated and applied.

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.

2 Related Work

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

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,

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

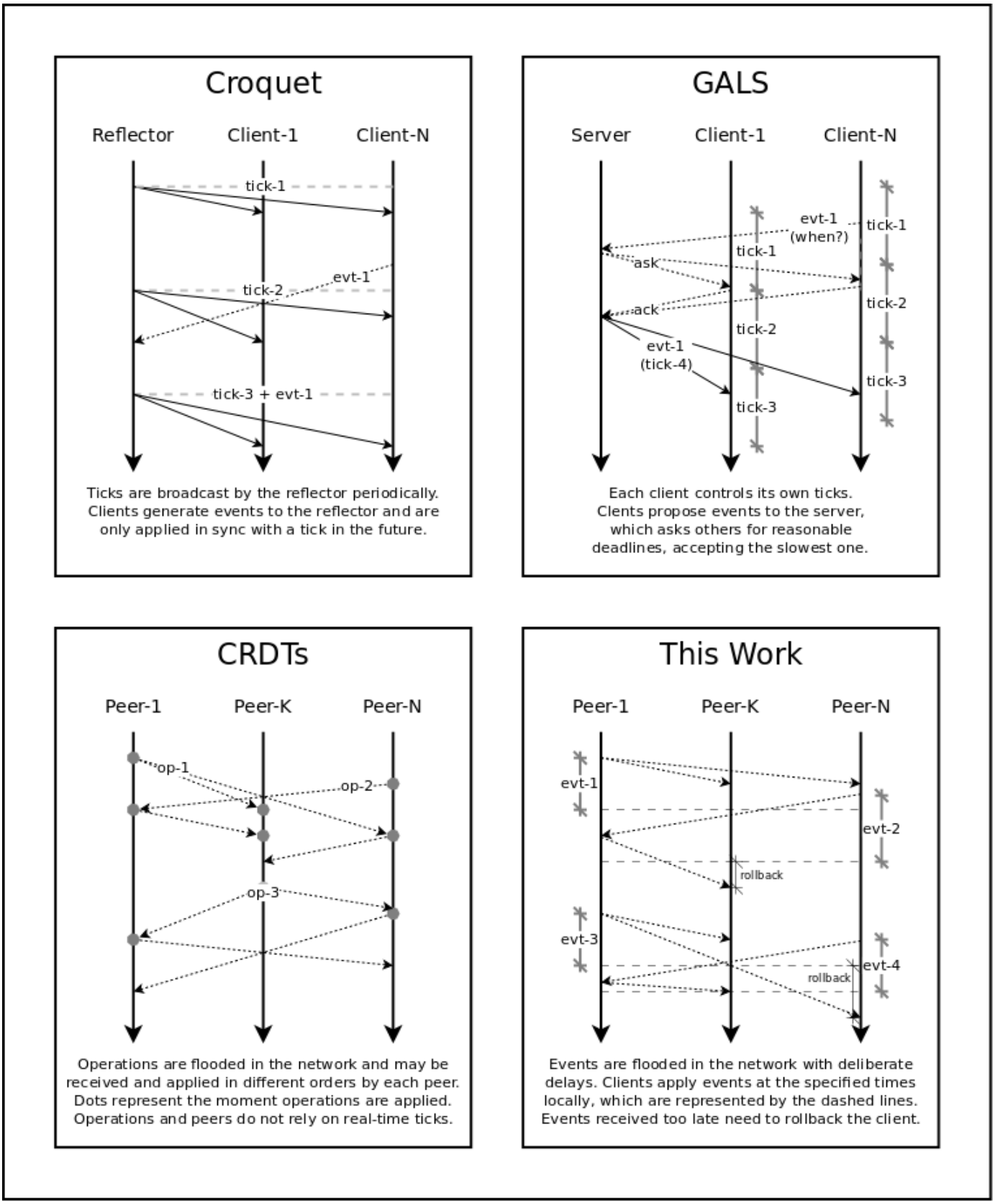

Fig. 2. Four approaches to symmetric distributed applications: Croquet, GALS, CRDTs, and this work. The vertical arrows in parallel represent the timelines in nodes in the network. The arrows crossing nodes represent communication between them.

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).

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 [3]. CRDTs are designed in such a way that all operations are commutative, so that the order in which they are applied does not affect the final outcome. As detailed in Figure 2, peers can communicate operations directly to each other with no central authority. Also, since operations need not to be ordered, they can be applied at the very first moment they are generated or received, even if the peers are offline. Since operations must be commutative, it is not possible to timestamp them in a 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 [4], since they can work offline in the same way as online. On the other hand, they provide very restrictive APIs (the CRDTs themselves), and do not support real-time consensus among peers. As indicated in Figure 1, CRDTs work on decentralized networks (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 [5], [6].

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. 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.

We now discuss related work on time machines. 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. 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 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 [7] proposes a simple API for time travelling in event-driven JavaScript: `init()` to return an initial state; `handle(evt,mem)` to process events and modify state; and `render(mem)` to output the current state. As we describe next, we use a similar approach by separating state modification from rendering [8].

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 under 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 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 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 satisfy. The

2. Fusion: https://doc.photonengine.com/en-us/fusion
3. Braid: http://braid-game.com/
4. SDL: http://www.libsdl.org
5. TODO

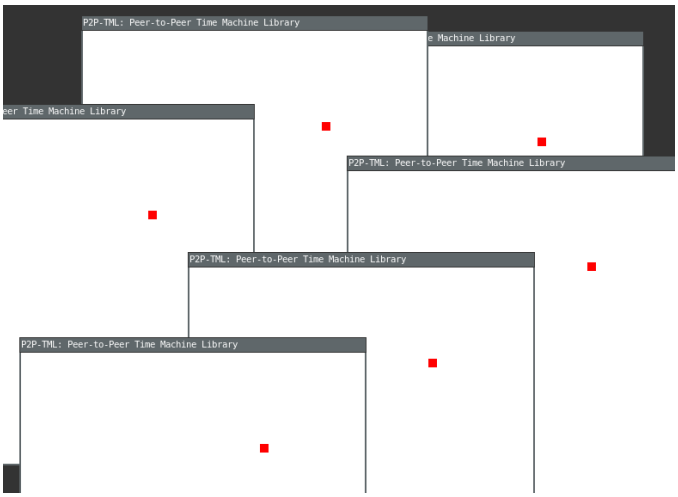

Fig. 3. A moving rectangle is controlled by peers connected indirectly. Users can press ↑ ↓ → ← to change the direction of the rectangle or `SPACE` to pause the application. The inputs are applied simultaneously in all peers, as if they were mirrors of a single application. Video: http://youtube.com/TODO

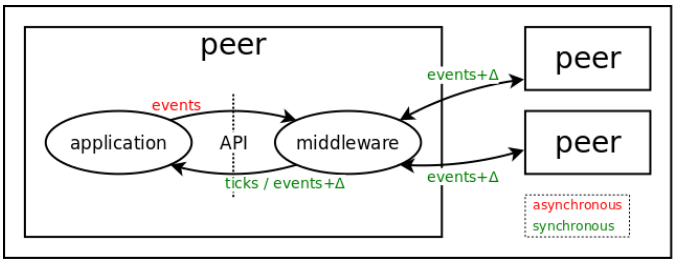

Fig. 4. Applications generate asynchronous events and communicate through an API with the middleware. The middleware orchestrates the peers, synchronizing their ticks locally and broadcasting events with a Δ delay.

middleware controls the execution of the application by issuing ticks and synchronized events.

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

a) **Event Deadlines:** Due to the inherent latency of networks, peers generate 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, 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, 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 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.

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

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 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 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` [7], which would otherwise render the screen multiple times while travelling. In our example, `cb_ren` just needs to redraw the rectangle at the current position:

```
void cb_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 broadcast application events, and (b) provide instant feedback to the user. Not all local low-level events need to broadcast 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 Broadcasting

As illustrated in Figure 4, events are broadcast between peers with a timestamp scheduled to the future with a Δ such that all peers are able to apply them in sync. To prevent broadcast 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 a broadcast 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. Therefore, this is not a severe limitation, and a separate mechanism could be used for assignments, with few modifications to the core of the middleware.

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 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        // 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*) (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 broadcast (*ln. 26–30*), also transforming it to 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.

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 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).

4 Evaluation

In this section, we evaluate if our design meets the goals to ensure that all peers (a) meet event deadlines, (b) advance

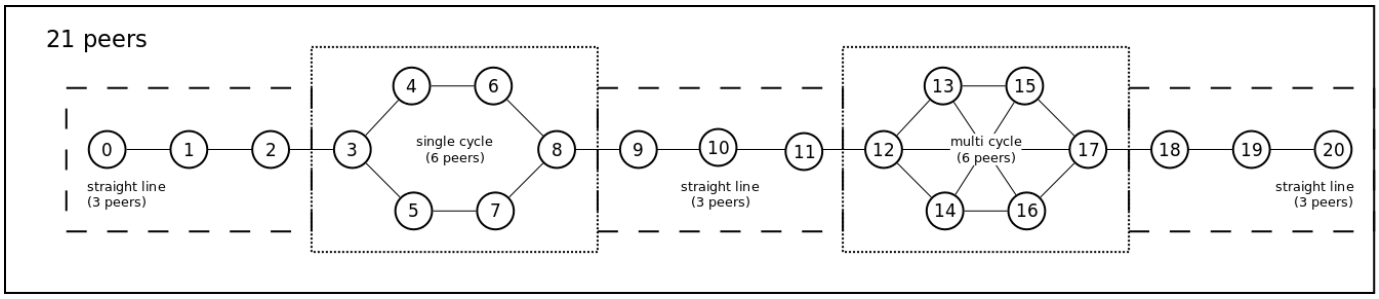

Fig. 5. Network topology: 21 peers organized as a straight line $(0-3)$, a simple cycle $(3-8)$, a second straight line $(8-11)$, a full connected cycle $(12-17)$, and a third straight line $(17-20)$.

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

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* [9] 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

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 $5x$, $2x$, 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*).

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 lantency 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.

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) 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 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 $50ms$, the peers need to roll back and resynchronize 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] D. A. Smith, A. Kay, A. Raab, and D. P. Reed, "Croquet-a collaboration system architecture," in *First Conference on Creating, Connecting and Collaborating Through Computing, 2003. C5 2003. Proceedings.* IEEE, 2003, pp. 2–9.

[3] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Symposium on Self-Stabilizing Systems*. Springer, 2011, pp. 386–400.

[4] M. Kleppmann, A. Wiggins, P. van Hardenberg, and M. McGranaghan, "Local-first software: you own your data, in spite of the cloud," in *Proceedings of Onward'19*, 2019, pp. 154–178.

[5] M. Kleppmann and A. R. Beresford, "A conflict-free replicated json datatype," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 2733–2746, 2017.

[6] P. van Hardenberg and M. Kleppmann, "Pushpin: Towards production-quality peer-to-peer collaboration," in *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, 2020, pp. 1–10.

[7] C. Schuster and C. Flanagan, "Live programming for event-based languages," in *Proceedings of the 2015 Reactive and Event-based Languages and Systems Workshop, REBLS*, vol. 15, 2015.

[8] S. Burckhardt, M. Fahndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato, "It's alive! continuous feedback in ui programming," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 95–104.

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