Symmetric Distributed Applications

Francisco Sant'Anna
francisco@ime.uerj.br
Rio de Janeiro State University (UERJ)
Brazil

Rodrigo Santos
rodsantos@microsoft.com
Microsoft
Brazil

Noemi Rodriguez
noemi@inf.puc-rio.br
PUC-Rio
Brazil

Abstract

A program is deterministic if multiple re-executions with the same inputs always lead to the same state. Even concurrent instances of a deterministic program should observe identical behavior—in real time—if assigned the same set of inputs. In this work, we propose real-time reproducibility for distributed programs. Multiple instances of the same interactive application can broadcast asynchronous inputs and yet conform to identical behavior. Collaborative networked applications, such as watch parties, document editing, and video games can benefit from this approach. We name this class of applications as *symmetric distributed applications*. Using a standard event-driven API to wait and emit events, programmers write code as if the application executes in a single machine. Our middleware intercepts event generation and synchronizes all instances in a consistent timeline so that receipt is identically reproducible. Not only distributed applications benefit from consistency and determinism but also development and testing can be done in a single instance with the same guarantees. In our experiments, the middleware can handle applications with 25 FPS, distributed in up to 25 nodes over the Internet, with an event latency below 350*ms*.

Keywords: consistency, determinism, GALS, synchronous programming, distributed applications

ACM Reference Format:
Francisco Sant'Anna, Rodrigo Santos, and Noemi Rodriguez. 2018. Symmetric Distributed Applications. In *REBLS '21: ACM Workshop on Reactive and Event-Based Languages and Systems, June 03–05, 2018, Chicago, IL.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/1122445.1122456

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
REBLS '21, June 03–05, 2018, Chicago, IL
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

1 Introduction

Deterministic programs are easier to understand, test, and verify [5]. Considering unpredictable user interactions, a program is deterministic if re-execution with the same order and timing of inputs always leads to the same state. With such timeline reproducibility property, multiple re-executions are indistinguishable from each other. Considering now distribution, it should be possible to provide a sequentially consistent timeline to concurrent instances of a deterministic program and also observe identical behavior *in real time*.

In this work, our goal is to support the real-time reproducibility property in a distributed setting. We propose that mirrored instances of the same application running in different machines should be able to broadcast asynchronous inputs to each other and yet conform to identical behavior. Hence, our focus is on *symmetric distributed applications*, instead of machines playing different roles in the network. A programmer writes the intended distributed application as if it would execute in a single machine. The underlying system we propose supports distribution and guarantees a sequentially consistent timeline to all instances, which react to this timeline deterministically. Hence, we support determinism locally at individual instances, and consistency globally at the whole distributed system level.

Collaborative networked applications fall in the class of symmetric distribution and can benefit from transparent reproducibility. As an example, *watch parties* are social gatherings to watch movies and TV shows. Users expect to be perfectly synchronized so that pressing the pause button in any machine should stop all others exactly in the same video frame. In this context, the delay imposed by the network is just an inconvenience that should not degrade the experience in comparison to users sitting in front of the same TV. Other examples that fall in this category are single-screen multiplayer games and collaborative document editing.

In addition to making distributed applications *behave* like local applications, we also intend to make distributed programs *be coded* like local programs. In this sense, we provide a standard event-driven API with two main commands to wait and emit events locally. We also provide the middleware that connects multiple application instances transparently in the network. The middleware intercepts local event generation and synchronizes all instances so that receipt is identically reproducible based on a consistent timeline. As a result, not only distributed applications benefit from consistency and determinism but also development and testing can be done in a single instance with the same guarantees. The middleware can handle applications with 25 FPS, distributed in up to 25 nodes over the Internet, with an event latency below 350*ms*.

As the main limitations, our middleware relies on a central server to determine a total order of events, and all instances must be known and responsive during the entire execution. In addition, since the latency of events is proportional to the maximum round-trip time (RTT) in the network, low-latency applications might become impractical. Finally, if clients diverge considerably from the expected RTT, applications may experience intermittent freezes. For these reasons, the proposed middleware targets soft real-time collaborative applications.

Section 2 describes the overall architecture of our middleware. Section 3 discusses the synchronous programming model, to which programs must comply to preserve determinism. Section 4 discusses how we adapted the globally-asynchronous locally-synchronous architecture with a synchronization protocol towards consistency and input reproducibility. Section 5 evaluates the performance and scalability of our middleware in multiple network and application scenarios. Section 6 compares our proposal with related work. Section 7 concludes this paper.

2 Overall Middleware Architecture

Figure 1 describes the client-server architecture of our middleware. A distributed application (*dapp*, at the top left of the Figure), is a set of mirrored instances running in different machines (also *dapp*, at the edges of the figure, to emphasize that they are symmetric and represent a single application). The clients are part of the middleware, but are co-located with each running *dapp* instance with very low communication latency. They intermediate all communication with the server and enable instances to be specified as local applications. As the figure illustrates, the latency between the instances and respective clients is in the order of microseconds, while the latency between clients and the server is in the order of milliseconds. The server receives asynchronous events (in red) from instances and redirects them to all clients as synchronous events with an appropriate delta delay on their timestamps (in green). The clients also control the local clock ticks in the instances and issue the received events at the appropriate timestamps (both synchronized, in green). Note that the clock ticks are not transmitted, nor synchronized, by the server. The timestamps represent the exact times at which all instances must apply the events to their identical local timelines. The delays in the events are necessary because of latency in network communications. Without delays, instances would inevitably receive events too late to apply them, with local times already ahead of event timestamps. In Section 4, we detail how to determine this delay in the synchronization protocol.

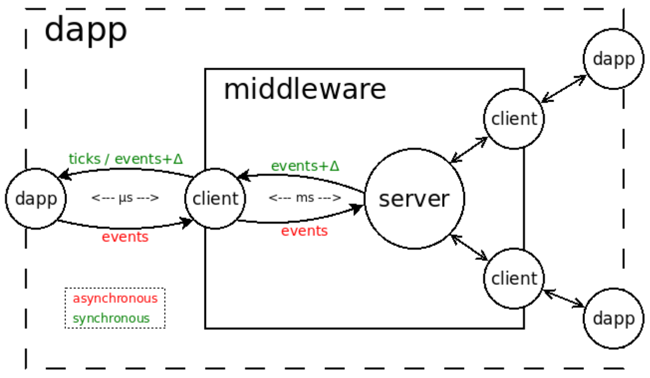


Figure 1. The architecture of the middleware. A single server synchronizes multiple clients, each connected to a mirrored instance of the distributed application.

Events represent user interactions, such as key presses, which are unpredictable and need to be communicated with the other instances. Since instances must be indistinguishable from each other by design, event sources are irrelevant and are ignored. For instance, if a user presses a key in one instance, the *dapp* behaves as if all users pressed the same key in all instances simultaneously. Clock ticks represent the rate at which applications advance, and are equivalent to frame rates in video playback and games. An important insight is that clock ticks are predictable inputs and need not go through the server. This results in no delay between the client and the *dapp* since local interprocess communication takes negligible time. Otherwise, unlike delays in sporadic mechanical user inputs, users would notice and condemn delays in clock ticks. Clock ticks and delayed events create a unique synchronous timeline shared by all instances of the *dapp*, making them manifest identical behavior. Figure 2 is an example of a timeline with asynchronous events that are synchronized with a delay. Except for id *0*, which represents clock ticks, applications determine their event ids, which the middleware just forwards with no further interpretation. As detailed in Section 4, the middleware ensures that all instances receive the same timeline.

Note that the delay between asynchronous event emission and corresponding synchronization is inherently nondeterminisitc due to the network. An hypothetical re-execution of an application with asynchronous emissions at the very same ticks would not guarantee that corresponding synchronizations would have the same delays. However, although a deterministic distributed timeline is impossible to achieve, our goal is to guarantee a consistent timeline to all instances. More specifically, we propose to extend Lamport's sequential consistency model [8] with a third property for timing accuracy as follows:

1. There is a total order of events in which all instances agree.

```
Tick    Event    Async
----    -----    -----
0000      0
0025      0       --> user presses key (1) and mouse (2)
0050      0
0075      0       --> user presses key (1)
0100      0
0125      1  <-- key is synchronized after delay
0150      0
0175      2  <-- mouse is synchronized after delay
0200      1  <-- key is synchronized after delay
0225      0
0250      0
....     ...
```

Figure 2. Example of a synchronized timeline of inputs shared by all instances of the *dapp*. Ticks are *25ms* apart (*40 FPS*). Asynchronous events are synchronized with a delay. Note that delay is unpredictable but event order within each source instance is preserved.

2. All events sent from a given instance are received in the same order by all others.
3. All instances receive the events at the same time.

We name this model *timely-sequential consistency* due to the third property.

The delta delay for user input is proportional to the maximum RTT considering all clients. We deliberately assume unbounded network delay to augment the possible application scenarios. Another concern is the rate of input generation in the instances. As an example, tracking the mouse position will inevitably flood the network with packets and make applications unresponsive. For these reasons, the feasibility of applications depends on (i) the acceptable delay in the user input, (ii) the nature and rate of inputs, and (iii) the maximum network latency.

The code for an actual *dapp* is the same for all instances and uses a standard event-driven API with only four commands:

- `mid_connect(port,fps)`:
  Connects with the local client in the given port and desired FPS.
- `mid_disconnect()`:
  Disconnects with the local client.
- `(now,evt) = mid_wait()`:
  Waits for the next input carrying a timestamp and event id.
- `mid_emit(evt)`:
  Emits an event corresponding to an asynchronous input.

Figure 3 shows the skeleton of a distributed application. The commands `connect` and `disconnect` are only required once to enclose the application logic (*ln. 2,10*).

Figure 2 with the timeline and 3 with the program relate as follows: The application initially blocks waiting for an input (*ln. 4*). According to the timeline, the first input happens at *tick 0* with *event 0* (no event). In the second iteration (*tick 25*), the application emits events *1* and *2* asynchronously (*ln. 8*). Only after a few ticks, these events are synchronized (*ticks 125 and 175*) in the main loop (*ln. 4*). As intended, note how the main event loop is coded like a standard local event-driven application. We extend this discussion in the next section.

```
01  fun dapp (port: Int) {
02     mid_connect(port,40)           // connects to client at 40 FPS
03     while (true) {                 // main event loop
04        val (now,evt) = mid_wait()  // awaits input (every 25ms)
05        switch (evt) {              // reacts to input
06           ...break loop...         //   possibly terminates
07        }
08        ...mid_emit(nxt)...         // possibly generates inputs
09     }
10     mid_disconnect()               // disconnects with the client
11  }
```

Figure 3. The skeleton of a *dapp* is a main loop that waits synchronous and emits asynchronous events.

3 Local Synchronous Programming

In the synchronous programming model [1], a program executes in locksteps (or logical ticks) as successive reactions to inputs provided by an external environment. In our context, the environment represents user interactions, and inputs can be occasional events, such as a key press, or simply the passage of time. Since execution is guided from outside, the main advantage of the synchronous model is that it is possible to record a sequence of inputs and reproduce the behavior of a program multiple times for reasoning and testing purposes. A fundamental requirement of synchronous programming, known as the *synchronous hypothesis* [11], is to isolate logical ticks from each other to preserve locksteps and prevent concurrent reactions to multiple inputs, which would break determinism. This hypothesis can be satisfied if computing reactions is faster than the rate of external inputs.

An important concern is how to guarantee that isolated reactions are themselves deterministic and sufficiently fast. Synchronous languages [2] typically restrict the programming primitives and/or perform static analysis to ensure these properties. However, since we propose a standard event-driven API targeting generic programming languages, we assume informally that the programmers themselves ensure these properties. This may involve coding best practices, such as avoiding preemptive multithreading, stateful system calls, and time-consuming loops in programs.

Figure 4 shows two common implementation schemes for synchronous programs [6]. In both schemes, a loop iteration updates the state of the memory completely before handling the next input. Hence, assuming that memory updates are deterministic, the only source of non-determinism resides in the order of inputs from the environment. Outputs are asynchronous events in the opposite direction of inputs and signal the environment about changes. They typically represent external actuators as opposed to input sensors.

```
< Initialize Memory >
foreach input_event do
    < Compute Outputs >
    < Update Memory >
end
```

a. "Event driven"

```
< Initialize Memory >
foreach period do
    < Read Inputs >
    < Compute Outputs >
    < Update Memory >
end
```

b. "Sampling"

Figure 4. Equivalent execution schemes for synchronous systems [6]: In the first scheme, an event occurrence triggers a loop iteration. In the second scheme, a predefined time interval triggers a loop iteration, which polls the environment for events.

The "Sampling" scheme of Figure 4 is very similar to the *dapp* skeleton of Figure 3: `mid_connect` specifies the sampling period (*ln. 2*), `mid_wait` reads inputs (*ln. 4*), `mid_emit` signals the environment back (*ln. 8*), and the `switch` statement processes the inputs to update memory (*ln. 5–7*). As required by the synchronous hypothesis, the sampling period, and consequently the rate of inputs, must be compatible with the processing speed of the loop body. A particularity of our design is that `mid_emit` is an output that corresponds to an asynchronous input that is later synchronized with a delay, as illustrated in Figure 1. The sampling scheme of Figure 4 is also adopted by popular event-driven libraries, such as *SDL* for computer graphics, and *Arduino* for embedded systems.[1] This allows for an easier integration of our middleware with practical systems, and also reinforces how, in our proposal, programming distributed versions becomes similar to their local counterparts.

Figure 5 is the code for a *dapp* written in SDL to control an animated rectangle on the screen with the keyboard.[2] The code structure follows the synchronous scheme with well delimited regions to initialize memory (*ln. 4–8*), read inputs in a loop (*ln. 11–13*), update memory (*ln. 15–28*), and compute outputs (*ln. 30–31*). As highlighted in the previous paragraph, outputs correspond to asynchronous inputs that are later retrofitted into the *dapp*. The output region is expanded in Figure 6 and is discussed next. The application first initializes the SDL library to create a window and renderer handle. The rectangle starts at position `(10,10)` with no movement in the axes: `vx=vy=0` (*ln. 7–8*). The main loop (*ln. 10–32*) first waits for the next input to control the rectangle (*ln. 11–13*). On each iteration, the screen is cleared and the rectangle is drawn on the current position (*ln. 15–19*). The `switch` statement (*ln. 20–27*) processes the current input: a clock tick (*ln. 21*) just moves the rectangle in the current direction; a space (*ln. 22*) pauses the rectangle by resetting the axes speeds; the arrow keys (*ln. 23–26*) set the axis speeds towards the appropriate direction. Before the next iteration, the screen is updated (*ln. 28*). The code is enclosed by middleware calls to connect and disconnect (*ln. 2,34*). Inside the main loop, all state modifications are sequential and depend only on the received event, which ensures that the application is responsive and deterministic.

```
01 int main (void) {
02   mid_connect(port, 40); // 25ms ticks
03
04   SDL_Init(SDL_INIT_VIDEO);                   ---\
05   SDL_Window*   win = SDL_CreateWindow();       |
06   SDL_Renderer* ren = SDL_CreateRenderer(win); |> Initialize
07   int x=10, vx=0; // position and               |    Memory
08   int y=10, vy=0; // speed multipler         ---/
09
10   while (1) {
11     uint64_t now;                             ---\
12     uint32_t evt;                               |> Read Inputs
13     mid_read(&now, &evt);                     ---/
14
15     SDL_SetRenderDrawColor(ren, WHITE);       ---\
16     SDL_RenderClear(ren); // clear screen       |
17     SDL_Rect r = { x, y, 10, 10 };              |
18     SDL_SetRenderDrawColor(ren, RED);           |
19     SDL_RenderFillRect(ren, &r); // draw rect   |
20     switch (evt) { // 5px/40fps -> 200 px/s     |
21       case 0:     { x+=5*vx; y+=5*vy; break; } |> Update Memory
22       case SPACE: { vy= 0; vx=0; break; }      |
23       case LEFT:  { vx=-1; vy=0; break; }      |
24       case RIGHT: { vx= 1; vy=0; break; }      |
25       case UP:    { vy=-1; vx=0; break; }      |
26       case DOWN:  { vy= 1; vx=0; break; }      |
27     }                                           |
28     SDL_RenderPresent(ren);                   ---/
29
30     // emit asynchronous inputs              ---\  Compute Outputs
31     ... mid_emit(evt) ...                     ---/
32   }
33
34   mid_disconnect();
35   return 0;
36 }
```

Figure 5. A *dapp* in SDL to control an animated rectangle on the screen with the keyboard.

Figure 6 expands the output region with calls to `mid_emit` that generate asynchronous inputs. In this application, the code simply calls the SDL library to check if a key is pressed to forward it to the middleware (*ln. 11–13*). Note that in a local-only application this region of code would replace the region to read inputs from the middleware. Note also that it is not necessary to customize this code for every single *dapp*. Instead, the middleware may provide a custom input generation stub for each event-driven library it supports, which *dapps* can reuse. This is the reason why we split this code in a separate figure.

To conclude this section, the "Update Memory" region of Figure 5 (*ln. 15–28*), which contains the core logic of the application, does not make calls to the middleware, being indistinguishable from a local version. In addition, since this region complies with the synchronous model premises, the application remains responsive and deterministic. For instance, the computations to update the positions are clearly faster than *25ms* (a tick period), satisfying the synchronous hypothesis. Also, if a timeline such as that of Figure 2 is applied to multiple re-executions of the application, then the

[1]SDL: www.libsdl.org. Arduino: www.arduino.cc.

[2]Video with two distributed instances: https://youtu.be/HSIwmrUenqg

```
// emit asynchronous inputs
SDL_Event inp;
if (SDL_PollEvent(&inp)) {
   if (inp.type == SDL_KEYDOWN) {
      switch (inp.key.keysym.sym) {
         case SDLK_LEFT:  { mid_emit(LEFT);  break; }
         case SDLK_RIGHT: { mid_emit(RIGHT); break; }
         case SDLK_UP:    { mid_emit(UP);    break; }
         case SDLK_DOWN:  { mid_emit(DOWN);  break; }
         case SDLK_SPACE: { mid_emit(SPACE); break; }
      }
   }
}
```

Figure 6. Asynchronous input generation with `mid_emit`. The library polls for key presses and forwards them to the middleware.

sequence of memory updates to `x`, `y`, `vx`, and `vy` (*ln. 20–27*) will be identical in every single execution. In the next section, we discuss how to extend these guarantees to a distributed setting.

4 Distributed GALS Architecture

The synchronous programming model assumes a single clock that controls the main loop to update the application state in locksteps. However, when we distribute the *dapp* in multiple instances, we can no longer assume a single clock source, which compromises the synchronous hypothesis. Nevertheless, towards our goal of symmetric distributed applications with identical behavior, we need to read perfectly synchronized inputs in real time in all instances. For instance, in the rectangle example of Figure 5, the challenge is to broadcast and then resynchronize all asynchronous calls to `mid_emit` (*ln. 31*), so that calls to `mid_read` (*ln. 13*) awake at the same time in all instances.

In the "Globally-Asynchronous Locally-Synchronous (GALS)" architecture, independent synchronous processes communicate asynchronously through the network [14]. Since processes are locally synchronous, asynchronous communication is the only source of nondeterminism. Inspired by the simplicity of GALS, we want not only to keep the local guarantees of synchrony, but also extend them globally, by transferring the responsibility of dealing with asynchrony to the middleware. As illustrated in Figure 1, *dapp* local instances depend on synchronous inputs only (green arrows), and are allowed to output asynchronous events (red arrows). However, unlike traditional GALS, the middleware automatically resynchronizes inputs in a single global timeline so that all instances observe them identically. Hence, the middleware must satisfy the timely-sequential consistency model for distributed timelines described in Section 2.

Resynchronization adds a delta delay to the original event that is proportional to the maximum RTT considering all clients. The reason for this is that the synchronization protocol needs to consider two main issues:

- **Clock differences:** Each instance has an independent clock that differs from others in *offset* and *rate*. The offset corresponds to the absolute point in time since the *dapp* started. As an example, suppose one instance started *5000ms* ago while another started *5051ms* ago. The *rate* corresponds to how fast the clock progresses, which may slightly differ between machines. As an example, suppose one instance misses *25ms* every hour in comparison to others. This difference, aka clock drift, may affect the *offset* considerably over time.
- **Network delay:** Instances are subject to communication delays with the server, which vary between clients and also over time. As an example, suppose a client next to the server has a stable RTT of *10ms*, while for a distant one, RTT may vary between *50–100ms* over time.

In our synchronization protocol, we do not assume any strict bounds in clock differences and network delays, which may of course affect the feasibility of some applications. The protocol has the following goals:

1. Ensuring that instances read inputs with perfect time accuracy.
2. Generating synchronous inputs back to instances with minimum delays.
3. Keeping clock offsets at instances close to each other.

Regarding *goal 3*, offset differences are inevitable but should be under a few milliseconds to be unnoticed by users. Regarding *goal 2*, the smaller the input delay, the better is the user experience regarding interactivity. However, if too small, a distant instance may receive the delayed input after its local time, which is unacceptable considering *goal 1*. In this case, the middleware may freeze the instance and pause its local time to ensure time accuracy, which is our ultimate goal.

It is important to recognize that an instance cannot by itself determine a reasonable timestamp for an event and broadcast it directly to others. First, the client has no idea about the network RTT to guarantee that all clients would receive the event in time to fulfill *goal 1*. Second, even if known, there is no guarantee that the RTT will not slightly increase momentarily, causing an instance to miss the event deadline. Third, even if the RTT was known and bounded, local instances might still diverge on local time offsets, again making an instance ahead in time to miss the deadline. Hence, our protocol relies on a central server to determine a reasonable timestamp for all clients, and to ensure that instances do not miss deadlines even if everything goes wrong.

Figure 7 describes the synchronization protocol performed by the middleware.[3] The next paragraphs explains the protocol in detail.

The server starts first and needs to know in advance how many instances will participate in the *dapp* (*Server (N)*). Each

[3]Video with two distributed instances, now with deliberate RTTs in the order of *100ms*: https://youtu.be/bf9C2mwkTzA

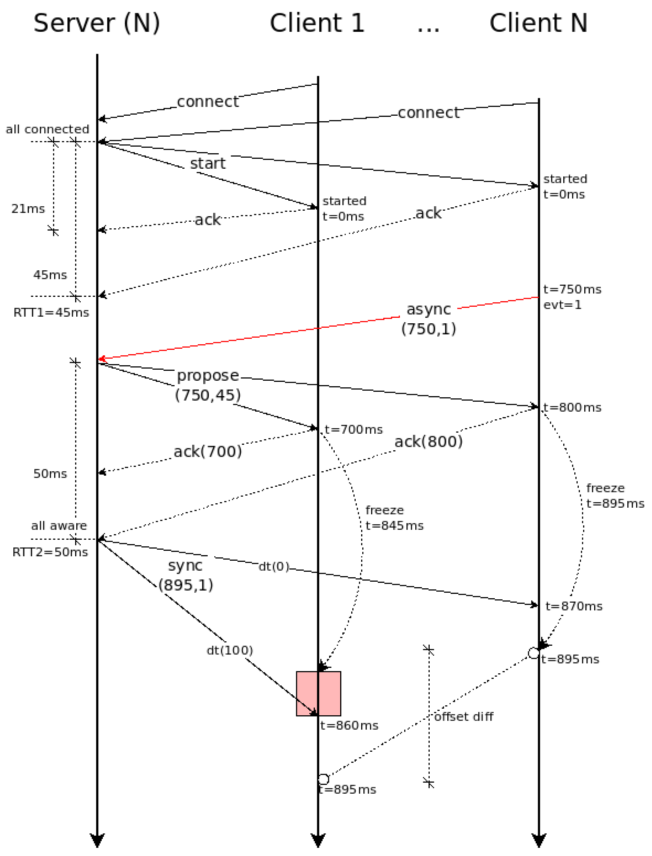


Figure 7. The synchronization protocol (i) keeps clock offsets together, (ii) generates synchronous inputs with minimum delays, and (iii) ensures that instances read inputs with accuracy.

instance connects through its client (*Client 1 ... Client N*) to the server, which in turn waits for all connections to succeed (*all connected*). Then, the server broadcasts a *start* message to all clients, which on receipt, start sending clock ticks to their instances (*started*). Note that clients will inevitably start at different absolute times due to network latency. The clients also send back an *ack* so that the server can calculate the maximum network RTT (*RTT1=45ms*), which is used next. After the startup process, each *dapp* instance executes at the same pace (except for drifts) with similar offset differences (at most *RTT1*). At this point, we consider instances to be indistinguishable from each other.

Now, let's consider an asynchronous event (in red). Since clients cannot determine a reasonable timestamp, *Client N* emits *evt=1* towards the server at local time *750ms* (*async(750,1)*). The server enqueues all requests and handles each event atomically, in sequence, as follows: First, it broadcasts the intention to emit an event (*propose(750,45)*), passing the source instance offset time (*750ms*) and current overall RTT (*RTT1=45ms*). Before it broadcasts the synchronous event with the final deadline (*sync(895,1)*), each client calculates its own deadline as follows: (i) takes the maximum time between the source and local offset, (ii) adds twice the received RTT, (iii) adds a small constant time (*5ms*), (iv) sets this sum as the local event deadline. As examples, the sum for *Client 1* is $max(750,700)+2x45+5 = 845$, and for *Client N* is $max(750,800) + 2x45 + 5 = 895$. This sum is reasonable because it takes into account the instance most ahead in time, and also the worst RTT with an extra margin. For instance, *Client N* with the worst RTT (*45ms*) is also ahead of time (*800ms*) and the final event will arrive after another round trip to the server, thus making *895ms* a reasonable deadline. The small constant depends on the number of instances in the application and is currently set to 200us per instance (e.g., *5ms* for *25* instances). During this process, the server also recalculates the RTT to use in the next event emission (*RTT2=50ms*).

Each client also starts a timer to freeze the instance if the (now expected) synchronous event is not received until its local deadline. In this case, the client pauses the instance by repeating the same clock tick over and over, until it receives the event. Although unfortunate, this at least ensures identical timelines in all instances to comply with *goal 1*. The server then waits for all client *acks* carrying their local offsets (*ack(700)* and *ack(800)*), recalculates each deadline (in the same way each client did), and takes the largest as the final event timestamp to broadcast to all clients (*sync(895,1)*). The server also sends drift compensations (*dt(100)* and *dt(0)*) based on offset differences between clients (we will discuss this later). At this point, all clients are expecting a timestamp that is greater than or equal to their own timer deadlines. Therefore, even if the network deteriorates, no instances will miss the synchronized event because they will be frozen before their deadlines.

Finally, the expected synchronous event is received by the clients (*sync(895,1)*). *Client N* receives it before the deadline, at local time *870ms*, requiring it to postpone local emission for another *25ms* to match the event timestamp with perfect accuracy (*895ms*). However, the timer at *Client 1* expires before receipt, requiring it to freeze the *dapp* for a while (red area). Then, the synchronous event is received at local time *860ms*, but still requires another *35ms* to match its timestamp. In the end, both clients emit the event exactly at local time *895ms* as expected (white circles).

Note that *Client 1* was harmed in the process mainly because its maximum time of *750ms* was much smaller than the *800ms* for *Client N*. This is because the local clocks were too distant from each other, breaking our *goal 3*. Note also that pausing *Client 1* increased the offset difference even further (*offset diff*). The middleware also implements an algorithm to compensate clock drifts as follows:[4] Once the server receives all acknowledgments carrying local offsets

[4]Video with two distributed instances, now with exaggerated clock drifts: https://youtu.be/wwlAURjN5YY

(*ack(700)* and *ack(800)*), the server sends back how much each client is behind in comparison to the maximum value (i.e., *800-700=100* and *800-800=0*). If this number is greater than zero, the client speeds up each local tick in *20%* until the time drift is compensated. For instance, if the client behind generates new ticks every *25ms*, it will instead generate *25ms* ticks every *20ms* for a while.

In order to maintain the overall network RTT updated and the clock offsets close to each other, the first client to connect with the server sends a periodic null event (*id=0*) every *5 seconds* to kick off the synchronization protocol automatically[5].

In the next section, we evaluate the performance of the protocol[6].

5 Evaluation

We present the protocol evaluation taking into consideration its main goals:

1. Ensuring that instances read inputs with perfect time accuracy.
2. Generating synchronous inputs back to instances with minimum delays.
3. Keeping instances clock offsets close to each other.

In order to verify if these goals are met, we created a series of experiments to measure the properties as follows:

- **Clock drift**: the percentage of drift compensation relative to the total simulation time (i.e., 1%, if drifts accumulate 1*s* in 100*s*).
- **Event latency**: the average number of frames elapsed from asynchronous emits to their corresponding synchronous receipts (e.g., 3, if an emit occurs at frame 58 and receipt at 61).
- **Frame freeze**: the percentage of repeated frames relative to the total simulation frames (i.e., 1%, if 1 frame freezes every 100 frames).

Drift compensation ensures that clock offsets are kept closer (*goal 3*), but a high percentage of clock drift may indicate that the protocol is actually causing it. Ideally, clock drift rate should be 0%. A high event latency may compromise *goal 2*, but a low latency may increase the amount of frame freezes, which is even less desirable. Ideally, event latency should be 1 frame, and freeze rate, 0%. Note that *goal 1* is a strict requirement, which the protocol ensures with 100% accuracy by design. In this sense, our experiments log the timelines of all instances and just asserts that they are identical in all simulations, since this is the only possible result.

We simulated a variety of scenarios with the following factors:

- *NET* (network connection type): affects network RTT.
- *FPS* (application frame rate): affects client CPU load.
- *N* (number of instances in the network): affects network traffic and server CPU load.
- *Rate* (rate of inputs): affects network traffic and client/server CPU load.

As an example of a concrete scenario, an interactive class over the Internet could have an RTT of *50–200ms* (*NET*), execute at *25* frames per second (*FPS*), with *50* instances (*N*), interacting every *5s* (*Rate*).

For each of these four factors, we tested the following values:

- *NET*: *Loopback*, *LAN*, and *WAN* (*0–50ms* RTT).
- *FPS*: 10, 25, 50, 100 frames per second.
- *N*: 2, 5, 10, 25, 50, and 100 instances.
- *Rate*: 5*s*, 1*s*, 500*ms*, and 250*ms* period between events.

These values result in 288 combinations. We executed each combination 3 times for 5 minutes each. The experiments take 72 hours to complete.

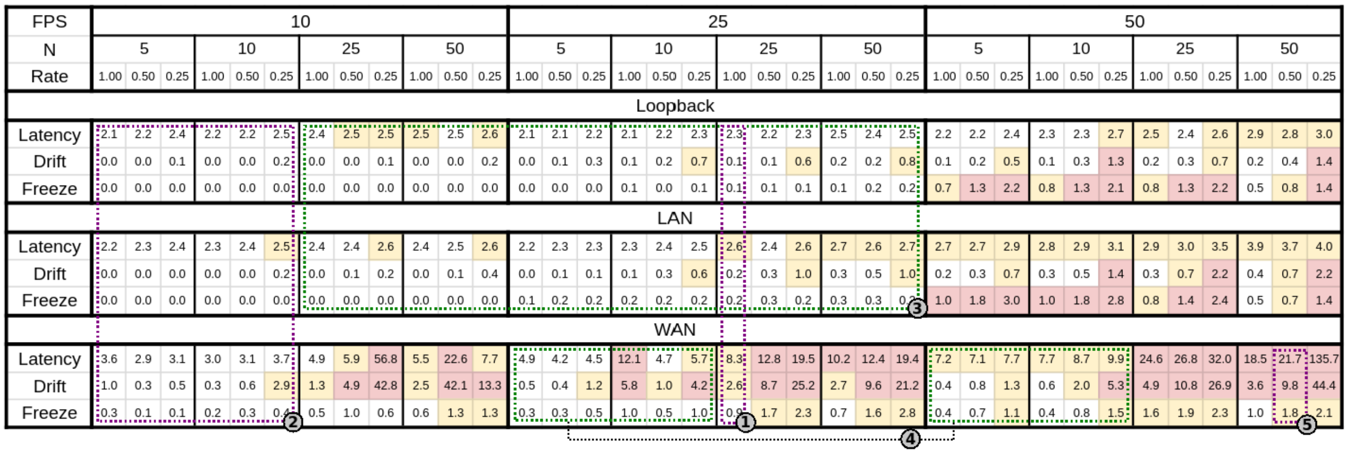
Figure 8 shows a summary with 108 out of all 288 measures, which are enough to draw the conclusions to follow. We omit *N=2*, *N=100*, *FPS=100*, and *Rate=5s* to save space in the figure. The color of the squares indicates their performances (white=good, yellow=moderate, and red=poor).

The thin vertical rectangle labelled **1** in the figure highlights a experiment running at 25 FPS, with 25 simultaneous instances, generating an event every second. The results show that this scenario is viable in all network configurations, including WAN: The latency around 8 frames means that an instance broadcasts an asynchronous event and applies it synchronized 330*ms* later, in the average. The drift of 2.6% means that for every minute of execution in each instance, the middleware needs to compensate 1500*ms*. The freezed frames of 0.9% means that for every 2400 frames (1 minute), 20 frames are repeated. Unsurprisingly, the results for local networks with the same factor levels are better, specially latency and drift.

Rectangle **2** indicates a safe region for all network types: up to 10 FPS, 10 instances, and 4 events per second. Rectangle **3** indicates a safe region for LANs: up to 25 FPS, 50 instances, and 4 events per second. The worst measured averages in these two large regions are a 2.7 latency in the LAN (110*ms*), a 2.9% drift and a 0.4% freeze over the WAN. Therefore, we can conclude that low frame-rate applications (10 FPS), with reasonable network size (10 instances) and high activity (4 events per second) can be reliable even over the WAN. For LANs, the reliability reaches high frame rates (50 FPS) and a larger network size (25 instances), which is enough even for video games in a local network.

Rectangles **4** indicate a region of interest for WANs: up to 50 FPS and 10 nodes. The worst measured averages in these rectangles are a 12.1 latency (500*ms*), a 5.8% drift, and a 1.5% freeze. This is a high frame rate region that admits a reasonable number of nodes, possibly with low event latency,

[5]Video with two distributed instances, now with random RTTs, clock drifts, and periodic synchronization: https://youtu.be/BYNHGsHnMx8
[6]Video with 100 distributed instances running in the Localhost: https://youtu.be/in5sdEEyGik



FPS	10											
N	5			10			25			50		
Rate	1.00	0.50	0.25	1.00	0.50	0.25	1.00	0.50	0.25	1.00	0.50	0.25
Loopback												
Latency	2.1	2.2	2.4	2.2	2.2	2.5	2.4	2.5	2.5	2.5	2.5	2.6
Drift	0.0	0.0	0.1	0.0	0.0	0.2	0.0	0.0	0.1	0.0	0.0	0.2
Freeze	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
LAN												
Latency	2.2	2.3	2.4	2.3	2.4	2.5	2.4	2.4	2.6	2.4	2.5	2.6
Drift	0.0	0.0	0.0	0.0	0.0	0.2	0.0	0.1	0.2	0.0	0.1	0.4
Freeze	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
WAN												
Latency	3.6	2.9	3.1	3.0	3.1	3.7	4.9	5.9	56.8	5.5	22.6	7.7
Drift	1.0	0.3	0.5	0.3	0.6	2.9	1.3	4.9	42.8	2.5	42.1	13.3
Freeze	0.3	0.1	0.1	0.2	0.3	0.4	0.5	1.0	0.6	0.6	1.3	1.3

FPS	25											
N	5			10			25			50		
Rate	1.00	0.50	0.25	1.00	0.50	0.25	1.00	0.50	0.25	1.00	0.50	0.25
Loopback												
Latency	2.1	2.1	2.2	2.1	2.2	2.3	2.3	2.2	2.3	2.5	2.4	2.5
Drift	0.0	0.1	0.3	0.1	0.2	0.7	0.1	0.1	0.6	0.2	0.2	0.8
Freeze	0.0	0.0	0.0	0.1	0.0	0.1	0.1	0.1	0.1	0.1	0.2	0.2
LAN												
Latency	2.2	2.3	2.3	2.3	2.4	2.5	2.6	2.4	2.6	2.7	2.6	2.7
Drift	0.0	0.1	0.1	0.1	0.3	0.6	0.2	0.3	1.0	0.3	0.5	1.0
Freeze	0.1	0.2	0.2	0.2	0.2	0.2	0.2	0.3	0.2	0.3	0.3	0.2
WAN												
Latency	4.9	4.2	4.5	12.1	4.7	5.7	8.3	12.8	19.5	10.2	12.4	19.4
Drift	0.5	0.4	1.2	5.8	1.0	4.2	2.6	8.7	25.2	2.7	9.6	21.2
Freeze	0.3	0.3	0.5	1.0	0.5	1.0	0.9	1.7	2.3	0.7	1.6	2.8

FPS	50											
N	5			10			25			50		
Rate	1.00	0.50	0.25	1.00	0.50	0.25	1.00	0.50	0.25	1.00	0.50	0.25
Loopback												
Latency	2.2	2.2	2.4	2.3	2.3	2.7	2.5	2.4	2.6	2.9	2.8	3.0
Drift	0.1	0.2	0.5	0.1	0.3	1.3	0.2	0.3	0.7	0.2	0.4	1.4
Freeze	0.7	1.3	2.2	0.8	1.3	2.1	0.8	1.3	2.2	0.5	0.8	1.4
LAN												
Latency	2.7	2.7	2.9	2.8	2.9	3.1	2.9	3.0	3.5	3.9	3.7	4.0
Drift	0.2	0.3	0.7	0.3	0.5	1.4	0.3	0.7	2.2	0.4	0.7	2.2
Freeze	1.0	1.8	3.0	1.0	1.8	2.8	0.8	1.4	2.4	0.5	0.7	1.4
WAN												
Latency	7.2	7.1	7.7	7.7	8.7	9.9	24.6	26.8	32.0	18.5	21.7	135.7
Drift	0.4	0.8	1.3	0.6	2.0	5.3	4.9	10.8	26.9	3.6	9.8	44.4
Freeze	0.4	0.7	1.1	0.4	0.8	1.5	1.6	1.9	2.3	1.0	1.8	2.1

Figure 8. The *Loopback*, *LAN*, and *WAN* regions each contains 36 measures for multiple configurations of *FPS*, *N*, and *Rate*. The rate of inputs appears in *seconds per events*, e.g., 0.50 means two events per second. The color thresholds (*yellow*,*red*) vary between WAN and local networks (LAN/Loopback) and are as follows: *Latency* (Local: 2.5,5 ; WAN: 5,10); *Drift* (Local: 0.5,1 ; WAN: 1,3); *Freeze* (Local: 0.5,1 ; WAN: 1,3). The *Loopback* experiments used a single machine with 1 process for the server, *N* processes for the clients, and *N* processes for the *dapp* instances. The *LAN* and *WAN* experiments used a machine dedicated for the server, and 3 other machines each with $N/3$ processes for the clients and instances. The experiments use heterogeneous *Linux* distributions with the machine configurations as follows: *Loopback* (*i7/8GB*); *LAN* (server *i7/8GB*; clients *i7/16GB*, *i7/8GB*, *i5/4GB*); *WAN* (server *Xeon-VM/1GB*; clients *2x i7/8GB*, *i5/4GB*).

which is enough for applications such as family watch parties over the Internet.

The other squares in the figure should be considered in isolation and depend on the target application and its requirements. As an example, let's consider rectangle **5**: 50 FPS, 50 instances, and 2 events per second for the WAN. A 21.7 latency corresponds to almost 1 second, but which might still be a reasonable delay for many applications. A drift of 9.8 means that the server will be constantly, but smoothly, compensating the instances clocks. A freeze below 2% over the Internet is also a reasonable value.

6 Related Work

Synchronous languages [2] focus on static guarantees for critical real-time control systems. They are based on a well-behaved lockstep execution that simplifies reasoning and verification of programs. They also restrict control and data primitives to enforce determinism and static bounds on memory usage and execution time. On the one hand, these guarantees enforce identical behavior to local instances in symmetric distributed application. On the other hand, the restrictive semantics of synchronous languages hinders the development of rich collaborative applications. We chose a pragmatic approach and provide a standard event-driven API with no restrictions. Nonetheless, this API can support host environments in synchronous languages.

In GALS architectures [14], computations within local synchronous nodes are deterministic, with communication latency being the only global source of nondeterminism. We propose a middleware to tame nondeterminism and provide global sequential consistency to all nodes. Some programming languages and systems [4, 9, 12] expose the GALS architecture to programmers, but which remain with the responsibility to coordinate interprocess communication over the network. Concurrent Reactive Processes (CRP) [4] extend the synchronous language Esterel [3] with asynchronous communication channels resembling CSP [7]. Esterel enforces determinism and bounded reactions to local instances, which are linked by asynchronous channels. Instances are asymmetric in the sense that they may execute different applications. Also, there is no central coordination towards a consistent global timeline, and each instance advances its clock independently. CRSM [12] and SystemJ [9] are similar systems, with CSP-like asynchronous channels and Esterel-like synchronous lockstep execution. The main difference to CRP is that they use Argos [10] (CRSM) and Java (SystemJ) as their local programming languages.

"Physically Asynchronous Logically Synchronous Systems (PALS)" [13] is a synchronization protocol to support strict real-time distributed applications. PALS provides a perfectly synchronized global clock shared by all instances, which leads to identical state transitions in all instances. On the one hand, PALS provides stronger synchrony in comparison to our protocol. In our proposal, local clocks advance independently and events need to be delayed to fit all instances timelines, possibly resulting in application freezes. On the other hand, PALS requires stronger assumptions, namely bounded limits for clock skews, network latency, and local computation time. We support unbounded limits that only affect the viability of applications in certain scenarios. Our protocol also keeps the instance clocks close to each other, but not as a strict real-time constraint. This is enough to keep the instances indistinguishable to the human eyes, and not affect the prediction of input delays. Regarding the protocol implementation, PALS also relies on a central server to serialize the timeline, but delivers a lower event latency. Since the bounds are fixed and known in advance, and all instances are at the same logical tick, PALS can skip the "proposal round" required by our protocol.

7 Conclusion

In this work, we propose *Symmetric Distributed Applications* as mirrored application instances that can broadcast inputs to each other and yet conform to identical behavior. Using a standard event-driven API, programmers write code as if the application executes in a single machine. Our middleware intercepts event generation and synchronizes all instances with a delay in a consistent timeline, so that receipt is identically reproducible. Not only distributed applications benefit from consistency and determinism but also development and testing can be done in a single instance with the same guarantees.

The middleware is based on a GALS architecture, in which nodes are locally deterministic, but subject to a global source of nondeterminism due to network latency. However, unlike traditional GALS, our middleware automatically resynchronizes inputs in a single global timeline so that all instances observe them identically. The middleware satisfies the timely-sequential consistency model, which extends sequential consistency with timing accuracy. The synchronization protocol works in two phases when receiving an asynchronous input: First, it asks the clients for a reasonable deadline, which also prepares them for the exact incoming timestamp. Next, the middleware collects all client responses and sends the synchronized event back to clients.

We evaluated the performance of the protocol with a series of experiments varying the network latency and size, and the application frame and event rate. We highlighted a reliable and high-performant region of parameters of up to 10 FPS and 10 instances for WANs, and of up to 25 FPS and 50 instances for LANs. We also identified a region of interest for WANs of up to 50 FPS and 10 nodes, but with an event latency in the order of $500ms$. The protocol evaluation helps identifying the distributed application scenarios that the middleware can handle.

References

[1] Albert Benveniste and Gérard Berry. 1991. The synchronous approach to reactive and real-time systems. *Proc. IEEE* 79, 9 (1991), 1270–1282.

[2] Albert Benveniste, Paul Caspi, Stephen A Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. 2003. The synchronous languages 12 years later. *Proc. IEEE* 91, 1 (2003), 64–83.

[3] Gérard Berry and Georges Gonthier. 1992. The Esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming* 19, 2 (1992), 87–152.

[4] Gérard Berry, S Ramesh, and RK Shyamasundar. 1993. Communicating reactive processes. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 85–98.

[5] Robert L Bocchino, Vikram Adve, Sarita Adve, and Marc Snir. 2009. Parallel programming must be deterministic by default. *Usenix HotPar* 6 (2009).

[6] Nicolas Halbwachs. 1998. Synchronous programming of reactive systems. In *International Conference on Computer Aided Verification*. Springer, 1–16.

[7] Charles Antony Richard Hoare. 1978. Communicating sequential processes. *Commun. ACM* 21, 8 (1978), 666–677.

[8] Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE transactions on computers* 28, 09 (1979), 690–691.

[9] Avinash Malik, Zoran Salcic, Partha S Roop, and Alain Girault. 2010. SystemJ: A GALS language for system level design. *Computer Languages, Systems & Structures* 36, 4 (2010), 317–344.

[10] Florence Maraninchi. 1991. The Argos language: Graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, Vol. 3. Citeseer.

[11] Dumitru Potop-Butucaru, Robert De Simone, and Jean-Pierre Talpin. 2005. The synchronous hypothesis and synchronous languages. *The embedded systems handbook* (2005), 1–21.

[12] S Ramesh. 1998. Communicating reactive state machines: Design, model and implementation. *IFAC Proceedings Volumes* 31, 32 (1998), 105–110.

[13] Lui Sha, Abdullah Al-Nayeem, Mu Sun, Jose Meseguer, and Peter C Olveczky. 2009. *PALS: Physically asynchronous logically synchronous systems.* Technical Report.

[14] Paul Teehan, Mark Greenstreet, and Guy Lemieux. 2007. A survey and taxonomy of GALS design styles. *IEEE Design & Test of Computers* 24, 5 (2007), 418–428.