Deterministic Distributed Interactive Applications

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

Rodrigo Santos
rodrim.c@gmail.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 guarantee real-time reproducibility for distributed programs. Multiple instances of the same interactive application are 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. 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 so that receipt is identically reproducible. Not only distributed applications benefit from determinism but also development and testing can be done in a single instance with the same guarantees.

Keywords: TODO

ACM Reference Format:
Francisco Sant'Anna, Rodrigo Santos, and Noemi Rodriguez. 2018. Deterministic Distributed Interactive 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, 6 pages. https://doi.org/10.1145/1122445.1122456

1 Introduction

Deterministic programs are easier to understand, test, and verify [3]. 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 reproducibility property, multiple re-executions are indistinguishable from one another. Considering now distribution, it should even be possible to provide the same set of inputs to concurrent instances of a deterministic program and observe identical behavior *in real time*.

In this work, our goal is to guarantee the real-time reproducibility property in a distributed setting. Mirrored instances of the same application running in different machines can 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.

Collaborative networked applications fall in the class of symmetric distribution and can benefit from transparent determinism and reproducibility. As an example, *watch parties* are social gatherings to watch movies and TV shows. Users expect to be perfectly synchronized such 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 make 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. Programmers write the intended distributed application as if it would execute in a single machine. We also provide the middleware to connect multiple application instances transparently in the network. The middleware intercepts event generation and synchronizes all instances so that receipt is identically reproducible. As a result, not only distributed applications benefit from determinism but also development and testing can be done in a single instance with the same guarantees.

As the main limitations, the middleware relies on a central server 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

REBLS '21, June 03–05, 2018, Chicago, IL

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

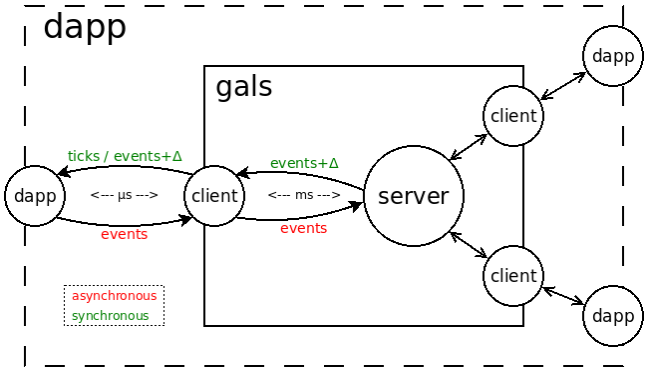

Figure 1. The architecture of the middleware *gals*. A single server synchronizes multiple clients, each connected to a mirrored instance of the distributed application.

```
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.

model, which programs must comply to preserve determinism. Section 4 discusses the globally-asynchronous locally-synchronous architecture of our middleware, and details the synchronization protocol for input reproducibility. Section 5 evaluates our middleware with... Section 6... Section 7...

2 Overall Middleware Architecture

Figure 1 describes the client-server architecture of our middleware entitled *gals*. 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, which are part of the middleware but co-located with each instance, intermediate all communication with the server and enable that *dapp* instances are specified as a local application. The server receives asynchronous events (in red) from instances and redirects them to all clients as synchronous events with an appropriate delta delay (in green). The delay is necessary because network communication takes non-negligible time (i.e., in the order of milliseconds), but instances need to advance together to preserve reproducibility. Hence, delaying the events is the only possibility to achieve reproducibility. The clients control the local clock ticks in the instances and issue the received events at the appropriate timestamps (both synchronized, in green).

Events represent user interactions, such as key presses, which are unpredictable and need to be communicated with the other instances. Since instances should be indistinguishable from each other, event sources are irrelevant. 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 in which applications are updated, and are equivalent to frame rates in video playback and games. We assume periods in the order of tens of milliseconds (e.g., 25 milliseconds or 40 frames per second). An important insight is that clock ticks are predictable inputs and need not to go through the server. This results in no delay between the client and the *dapp* since interprocess communication takes negligible time (i.e., in the order of a few microseconds). Unlike for sporadic mechanical user inputs, users would notice and condemn delays in clock ticks. Clock ticks and delayed events constitute the 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 event id *0*, which represent clock ticks, applications determine their own ids. The middleware just forwards events with no further interpretation. As detailed in Section 4, the middleware ensures that all instances receive the same timeline.

The delta delay for user input is proportional to the maximum network round-trip time 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 the application unresponsive. For these reasons, the viability 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:

- `connect(port,fps)`:
  Connects with the local client in the given port and desired FPS.
- `disconnect()`:
  Disconnects with the local client.
- `(now,evt) = gals_wait()`:
  Waits for the next input carrying a timestamp and event id.

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

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

- `gals_emit(evt)`:
  Emits the given 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.

Figure 2 and 3 with the timeline and 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.

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* [5], is to isolate logical ticks from each other to preserve locksteps and prevent concurrent reactions to 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 our solution proposes a standard event-driven API targeting generic programming languages, we assume these properties are ensured informally. This may involve coding best practices, such as avoiding stateful system calls and time-consuming loops.

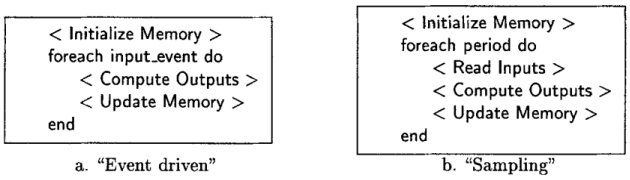

Figure 4. Equivalent execution schemes for synchronous systems [4]: In the first scheme, an event occurrence triggers a loop iteration. In the second scheme, a predefined time interval triggers a loop iteration, which polls the environment for events.

Figure 4 shows two common implementation schemes for synchronous programs [4]. In both schemes, a loop iteration updates the state of the memory completely before handling the next input. Hence, assuming 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.

The "Sampling" scheme of Figure 4 is very similar to the *dapp* skeleton of Figure 3: `gals_connect` specifies the sampling period (*ln. 2*), `gals_wait` reads inputs (*ln. 4*), `gals_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. A subtle particularity is that *dapps* outputs are actually asynchronous inputs later retrofitted back synchronized with a delay, as described in Figure 1. The sampling scheme 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. The code structure follows the synchronous implementation 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 mentioned above, the outputs are actually asynchronous inputs that are later retrofitted into the *dapp*. The output region is expanded in Figure 6 and is discussed in sequence. The application first initializes the SDL library to create a window and renderer handle (*ln. 4–6*). 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

[1] www.libsdl.org and www.arduino.cc

```
01  int main (void) {
02    gals_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      gals_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      ... gals_emit(evt) ...                  ---/
32    }
33
34    gals_disconnect();
35    return 0;
36  }
```

Figure 5. A *dapp* in SDL to control an animated rectangle on the screen with the keyboard. Video with two running instances: TODO

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*) sets 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 depend only on the received event, which ensures that the application is deterministic.

Figure 6 expands the output region with the calls to `gals_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. 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 application. 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

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

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

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, the sequence of memory updates to `x`, `y`, `vx`, and `vy` (*ln. 20–27*) will be identical every single time. 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 introduce a delay due to communication, and we can no longer assume a single clock source, which compromises the synchronous hypothesis.

Towards symmetric distributed applications with identical behavior, we need to read inputs synchronized and in real time in all instances. For instance, in the rectangle example of Figure 5, the challenge is to awake calls to `gals_read` at the same time in all instances (*ln. 13*), and to broadcast and then resynchronize all asynchronous calls to `gals_emit` (*ln. 31*).

The "Globally-Asynchronous Locally-Synchronous Architecture (GALS)" integrates multiple independent synchronous processes as a single distributed application [**?**]. By adopting this architecture, we can keep the simplicity and guarantees of the synchronous model to program local instances, and transfer the responsibility to deal with asynchrony to a middleware at the global level. In Figure 1, we propose this architecture. We can see that *dapp* local instances depend on synchronous inputs only (green arrows), and are allowed to output asynchronous events (red arrows), whose responsibility to resynchronize is left to the middleware.

As introduced in Section 2, resynchronization adds a delta delay to the original event proportional to the maximum network round-trip time considering all clients. The middleware synchronization algorithm needs to consider two main problems:

- **Clock differences:** Each instance has an independent clock that differs from others in *offset* and *rate*. The offset refers to the point in time since the *dapp* started. As an example, suppose one instance started *5000ms* ago while another started *5051ms* ago. The *rate* refers 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 time offset considerably over time.
- **Network delay:** Instances are subject to a communication delay with the server, which varies between clients and also over time. As an example, suppose a client next to the server has a consistent RTT of *10ms*, while a distant one may vary between *50–100ms* over time.

We do not assume any strict timing bounds in our synchronization algorithm, which may of course affect the viability of some applications.

Overall, the synchronization algorithm performed by the middleware has the following goals:

1. Keep clock offsets in instances close to each other.
2. Generate synchronous inputs to instances with a minimum delay.
3. Ensure that instances read inputs with perfect time accuracy.

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

Figure 7 describes the synchronization protocol performed by the middleware. 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 instance connects through its client with the server (*Client 1 ... Client N*), which in turn waits for all connections to succeed (*all connected*). Then, the server broadcasts a *start* message to all clients, which on receipt, starts 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

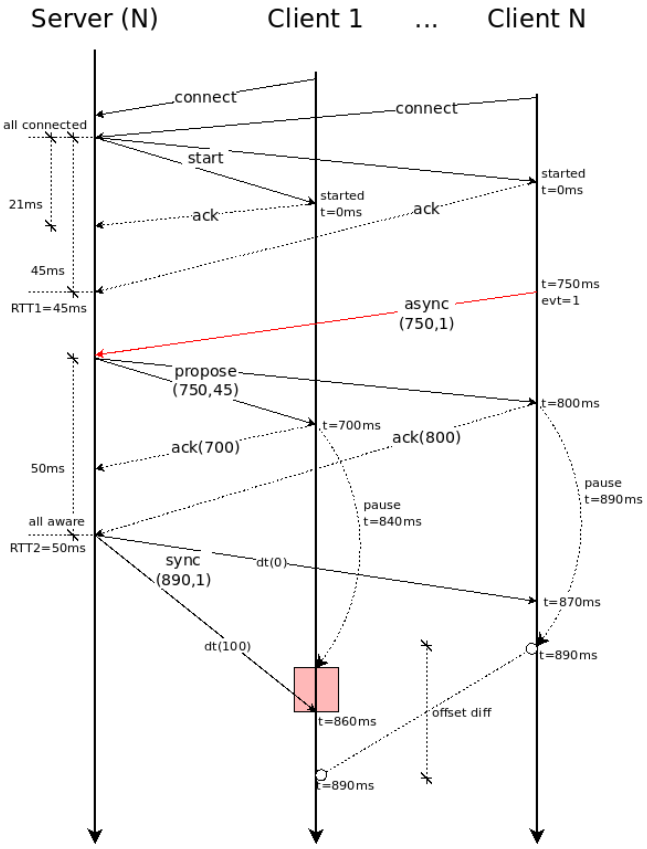

Figure 7. The synchronization protocol (i) keeps clock offsets together, (ii) generates synchronous inputs with minimum delays, and (iii) ensures that instances read inputs with accuracy.

executes at the same pace (except for drifts) with similar offset differences (at most *RTT1*). We consider that, at this point, the instances are indistinguishable from each other.

Now, let's consider an asynchronous event (in red): *Client N* emits event *1* at local time *750ms* (*async(750,1)*). The server enqueues all event requests and handles each one atomically, in sequence, as described further. It is important to recognize that a client 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 propose a deadline timestamp that guarantees that all clients receive the event in time to fulfill *goal 3* (e.g, *750+RTT*). Second, even if known, there is no guarantee that the RTT will not slightly increase momentarily, making 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 (e.g., *750ms* vs *800ms*), again making an instance ahead to miss the deadline. As mentioned above, we do not assume any bounds in RTT and offset variance.

Hence, our protocol relies on the server to determine a reasonable timestamp for all clients, and to ensure that

instances do not miss deadlines even if everything goes wrong. First, the server broadcasts the intention to emit an event (*propose(750,45)*), passing the source instance offset time (*750ms*) and current overall network RTT (*RTT1=45ms*). The final synchronous event (*sync(890,1)*) is yet to come after all clients acknowledge the server. Before that, each client calculates its own deadline as follows: (i) take the maximum time between the source and local offset, (ii) add twice the received RTT to this maximum, and (iii) set this sum as the local event deadline. As examples, the sum for *Client 1* is $max(750, 700) + 2x45 = 840$, and for *Client N* is $max(750, 800) + 2x45 = 890$. This sum is reasonable because it considers 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 (*event(890,1)*), thus making *890ms* a reasonable deadline. 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 pause the instance if the (now expected) synchronous event is not received until the 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 of the *dapp*, ensuring *goal 3*. 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(890,1)*). The server also sends drift compensations (*dt(100)* and *dt(0)*) based on offset differences between clients, which is discussed further. 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 paused before its deadline.

Finally, the expected synchronous event is received by the clients (*sync(890,1)*). *Client N* receives it before the deadline, at local time *870ms*, which requires to postpone local emission for another *20ms* to match the event timestamp with perfect accuracy (*890ms*). However, *Client 1* expires its timer before receipt and needs to pause the *dapp* for a while (red area). Then, the synchronous event is received at local time *860ms*, but still requires another *30ms* to match its timestamp. In the end, both clients emit the event exactly at local time *890ms* 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 Client N* had. This is because the local clocks were too distant from each other, breaking our *goal 1*. As an additional remark, note that pausing *Client 1*, increased the offset difference even further (*offset diff*). The middleware also implements an algorithm to compensate clock drifts as follows: Once the server receives all acknowledges carrying local offsets (*ack(700)* and *ack(800)*), the server sends back how much each client is delayed 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*.

5 Evaluation

6 Related Work

7 Conclusion

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

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

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