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 order of inputs always lead to the same state. For a given deterministic program, it should even be possible to provide the same set of inputs to concurrent instances and observe identical behavior in real time. In this work, we guarantee real-time reproducibility in a distributed setting. Multiple instances of the same 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 of 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 of 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, 4 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 of 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 anyone pressing the pause button should stop all instances exactly in the same video frame. In this context, the network distributions is just an inconvenience that should not make the experience to diverge from users sitting in front of the same TV. Other examples that fall in this category are single-screen multiplayer games and collaborative document editing.

Since our goal is to make distributed applications to *behave* like local applications, we also intend to make distributed programs to *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 that connects the 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 centralized server and all instances must be known and responsive during the entire execution. In addition, the latency of events is the maximum round-trip time (RTT) considering all clients, which can be intolerable for low-latency applications such as video games. Finally, if a client diverges from the expected RTT, the application 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

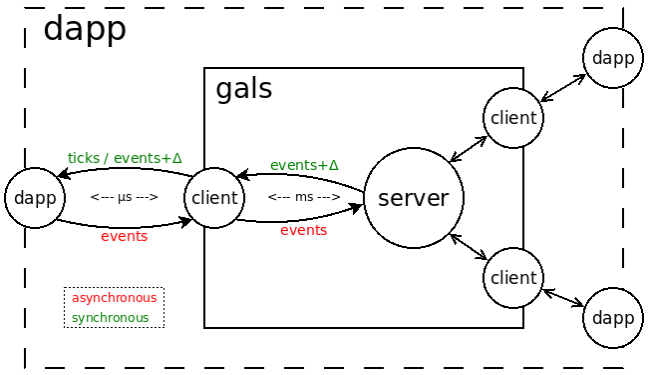

Figure 1. The architecture of the middleware *gals*. A single server synchronizes multiple clients, each connected to a mirrored instance of the distributed application.

```
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 instance 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 algorithm for real-time distributed 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*, inside the circles, 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 is key to permit that *dapp* instances are specified as a local application. The server receives asynchronous events (in red) from instances and is responsible for redirecting them to all clients as synchronous events with an appropriate delta delay (in green). The delay is necessary because network broadcast takes non-negligible time (i.e., in the order of milliseconds), and instances need to advance at the same time in order to preserve reproducibility. The clients control the clock ticks of the instances and is responsible for issuing the received events at the appropriate timestamps (both synchronized, in green).

The 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 one another, 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. For practical purposes, 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 *dapp* and the client since interprocess communication takes negligible time (i.e., in the order of a few microseconds). Unlike sporadic mechanical user input, clock ticks cannot tolerate considerable delays without going unnoticed by humans. Clock ticks and delayed events constitute the unique synchronous timeline shared by all instances of the *dapp*, allowing them to manifest identical behavior. Figure 2 is an example of a timeline with asynchronous events that are synchronized with a delay. Except for inputs with event id *0*, which reprepresent clock ticks, each application determines its own ids, and the middleware just forwards them with no processing. As detailed in Section 4, the middleware ensures that all instances receive the same timeline.

The delta delay for user input is the maximum network round-trip time considering all clients. We deliberately assume unbounded network delay to augment the scope of applications. 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           ...gals_emit(nxt)...     //   possibly generates inputs
07           ...break loop...         //   possibly terminates
08        }
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 an 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 related as follows: The application initially blocks at *line 4* waiting for an input. According to the timeline, the first input happens at *tick 0* and *event 0* (no event). In the second iteration *tick 25*, the application emits events *1* and *2* asynchronously at *line 6*. Only after a few ticks, these events are synchronized and awake *line 4* in subsequent ticks. The main event loop is coded like a standard local event-driven application as intended.

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 can record a sequence of inputs and reproduce the behavior of a program multiple times for reasoning and testing purposes. A fundamental requirement for synchronous programming, known as the *synchronous hypothesis* [5], is to isolate logical ticks from one another to preserve locksteps and prevent concurrent reactions to inputs, which would break determinism. This hypothesis is 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 like avoiding stateful system calls and time-consuming loops.

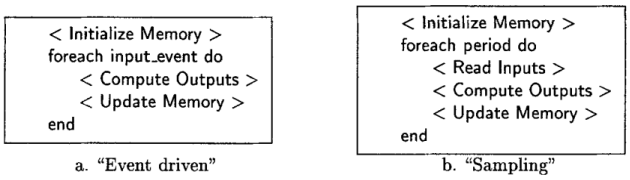

Figure 4. Equivalent execution schemes for synchronous systems. [4] In the first scheme, a change in the environment is designated as an input event. In the second scheme, an instant is a predefined time interval in which the environment is polled for input events.

Figure 4 shows two common implementation schemes for synchronous systems [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, `gals_wait` reads inputs, `gals_emit` signals the environment back, and the `switch` statement processes the inputs (to update memory). 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 programming distributed versions are 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*. This region of code 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, it simply calls the SDL library to check if a key is pressed and forward 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, and *dapps* may reuse it. This is the reason why we split this code in a separate figure.

To conclude this section, the "Update Memory" region of Figure 5, which contains the core logic of the application,

```
// 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.

does not make calls to the middleware, being indistinguishable from a local version. In addition, if this region complies with the synchronous model premises, the application remains responsive and deterministic.

4 Distributed GALS Architecture

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.