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. Mirrored instances of the same application are allowed to 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 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.

Section 2 describes the overall architecture of our middleware. Section 3 discusses the synchronous programming model, which programs following our event-driven API must comply in order to preserve determinism. Section 4 discusses the globally-asynchronous locally-synchronous architecture

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

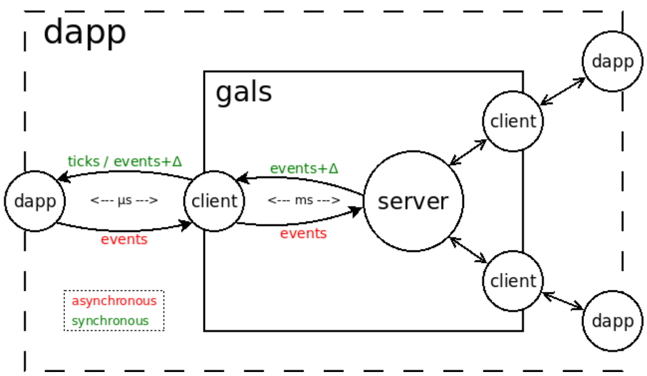

Figure 1. The architecture of the middleware *gals*. A single server synchronizes multiple clients, each connected to a mirrored instance of the distributed application.

```
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.

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, 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. 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.
- `gals_emit(evt)`:
  Emits the given asynchronous input.

Figure 5 shows the skeleton of an application. The commands `connect` and `disconnect` are only required once to enclose the application logic.

```
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.

```
< 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. [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 2 and 5 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 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 events in the opposite direction of inputs and signal the environment about changes. They typically represent actuators (as opposed to input sensors), such as the screen or printer.

The "Sampling" scheme of Figure 4 is very similar to the *dapp* skeleton of Figure 5: `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 a subtle difference, instead of output events, *dapps* emit asynchronous inputs that are later retrofitted back as synchronous inputs with a delay, as described in Figure 1. As required by the synchronous hypothesis, the sampling period, and consequently the rate of inputs, must be compatible with the processing speed.

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 **??** to move a rectangle in the screen

- SDL stub would allow for ...
how development is similar.
for interaction with
... TODO ...

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).

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

```
int main (void) {
  gals_connect(port, 40); // 25ms ticks

  SDL_Init(SDL_INIT_VIDEO);                       \
  SDL_Window*   win = SDL_CreateWindow("dapp");   |
  SDL_Renderer* ren = SDL_CreateRenderer(win);    |> Initialize
  int x=10, vx=0; // position and                 |    Memory
  int y=10, vy=0; // speed multipler              /

  while (1) {
    uint64_t now;                                 \
    uint32_t evt;                                 |> Read Inputs
    gals_read(&now, &evt);                        /

    SDL_SetRenderDrawColor(ren, WHITE);           \
    SDL_RenderClear(ren); // clear screen         |
    SDL_Rect r = { x, y, 10, 10 };                |
    SDL_SetRenderDrawColor(ren, RED);             |
    SDL_RenderFillRect(ren, &r); // draw rect     |
    switch (evt) { // 5px/40fps -> 200 px/s       |> Update Memory
      case 0: { x+=5*vx; y+=5*vy; break; }        |
      case 1: { vx=-1; vy=0; break; }             |
      case 2: { vx= 1; vy=0; break; }             |
      case 3: { vy=-1; vx=0; break; }             |
      case 4: { vy= 1; vx=0; break; }             |
      case 5: { vy= 0; vx=0; break; }             |
    }                                             |
    SDL_RenderPresent(ren);                       /

    // emit asynchronous inputs                   \  Compute Outputs
    ... gals_emit(evt) ...                        /
  }

  gals_disconnect();
  return 0;
}
```

Figure 5. The skeleton of a *dapp* is a main loop that waits synchronous and emits asynchronous events.

[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.