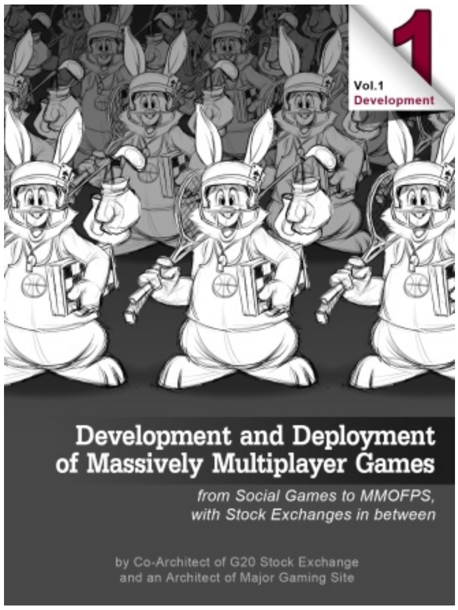
Chapter V(c). Modular Architecture: Client-Side. On Debugging Distributed Systems, Deterministic Logic, and Finite State Machines

posted December 7, 2015 by "No Bugs" Hare, translated by Sergey Ignatchenko

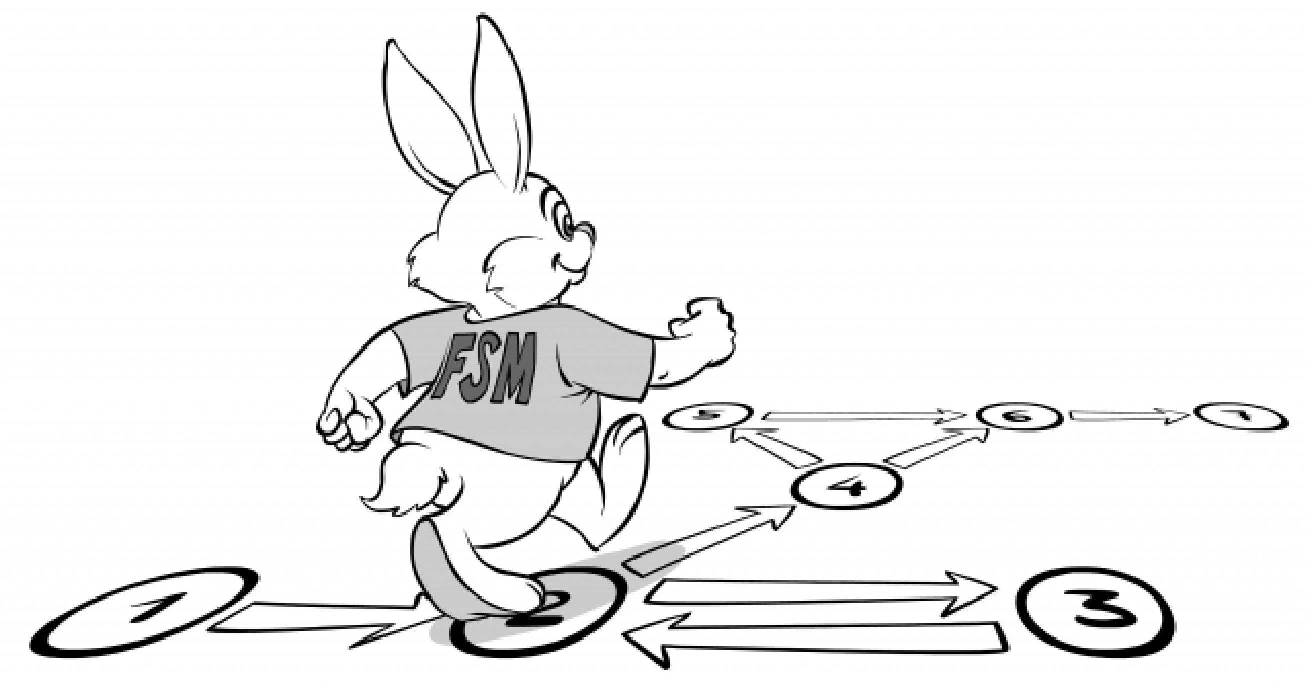
[[*This is Chapter V(c) from the upcoming book "Development&Deployment of Massively Multiplayer Online Games", which is currently being beta-tested. Beta-testing is intended to improve the quality of the book, and provides free e-copy of the "release" book to those who help with improving; for further details see "Book Beta Testing". All the content published during Beta Testing, is subject to change before the book is published.*



To navigate through the book, you may want to use Development&Deployment of MMOG: Table of Contents.]]

Distributed Systems: Debugging Nightmare

Any MMOG is a distributed system by design (hey, we do need to have a server and at least a few thousands of clients). While distributed systems tend to differ from non-distributed ones in quite a few ways, one aspect of distributed systems is especially annoying. It is related to debugging.



The problem with debugging of distributed systems is that it is usually impossible to predict all the scenarios which can happen in real world. In short, we're speaking about race conditions. While it is usually possible to answer "What will happen if such a packet arrives at exactly such and such moment", making an *exhaustive* list of such questions is unfeasible for any distributed system which is more complicated than a stateless HTTP request-response "Hello, Network". If you didn't try creating such an *exhaustive* list yourself for a non-trivial system – you may want to try doing it, but it will be much cheaper to believe my experience in this field – for any non-trivial stateful system it won't work, period.

This automatically means that even the best possible unit testing (while still being useful) inevitably fails to provide any guarantees for a distributed system. Which in turn means that in many cases you won't be able to see the problem until it happens in simulation testing, or even in real world. To make things even worse, in simulation testing it will happen every time at a different point. And when it happens in real world, you usually won't be able to reproduce it in-house. Sounds grim, right? It certainly does, and for a reason too.

As a result, I am going to make the following quite bold statement:

Race condition

A race condition or race hazard is the behavior of an electronic, software or other system where the output is dependent on the sequence or timing of other uncontrollable events.

— *Wikipedia* —

If you don't design your distributed system for testing and post-mortem analysis,[1] you will find yourself in lots of trouble

Fortunately, it *is* possible to design your system for distributed testing, and it is *not* that difficult, but it requires certain discipline and is better to be done from the very beginning; we'll discuss one of the ways to do it a little bit later.

The Holy Grail of Post Mortem

In practice, whatever amount of testing you do, test cases produced by real life and by your inventive players, will inevitably go far beyond everything you were able to envision in your tests. It means that from time to time your program will fail. While reducing time between failures is very important, another thing is arguably even more important than that: it is the time which takes you to fix the bug after it was reported. And for reducing number of times which the program needs to fail before you can fix it, post-mortem analysis is of paramount importance. The holy grail of post-mortem, of course, is when you can fix any bug using the data from one single crash, so it doesn't affect anybody anymore. This holy grail (as well as any other holy grail) is not achievable in practice. However,

I've seen systems which, using techniques similar to those described in this book, were able to fix around 75% of all the bugs after a single post-mortem

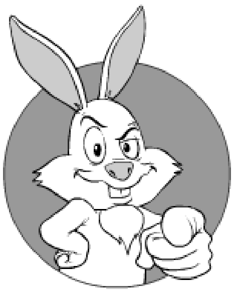
---

[1] here we're speaking about post-mortem analysis after program failure, core dump or otherwise, and *not* about "project post-mortem"

Portability: Platform-Independent Logic as "Nothing But Moving Bits Around"

Now let's set aside all the debugging for a moment, and speak a little bit about platform independent stuff. Yes, I know I am jumping to quite a different subject, but we do need it, you will see how portability is related to debugging, just half a page later.

In most cases graphics, input, and network APIs on different platforms will be different. Even if all your current platforms happen to have the same API for one of the purposes, chances are that your next platform will be different in this regard.



"It is necessary to separate your code into two very-well-defined parts: platform-dependent one and platform-independent one.

As a result, it is necessary to separate your code into two very-well-defined parts: platform-dependent one and platform-independent one. Moreover, similar to what we've discussed with regards to Logic-to-Graphics Layer (see "Logic-to-Graphics Layer" section above), your platform-dependent code needs to be very-rarely-changing, and your frequently-changing game logic needs to be platform-independent.

When speaking about platform-independent logic, a friend and colleague of mine, Dmytro Ivanchykhin, likes to describe it as "nothing more then moving bits around". Actually, this is quite an accurate description. If you can isolate a portion of your program in such a way that it can be described as mere "taking some bunches of bits, processing them, and giving some other bunches of bits back", all of this while making only those external calls which are 100% cross-platform (ok, "100% cross-platform for all the platforms you need"), you've got your logic platform-independent.

Having your game logic on the client side as a platform-independent logic, is absolutely necessary for any kind of cross-platform development. There is no way around it, period; any attempts to have your game logic interspersed with the platform-dependent calls will doom your code sooner rather than later. This is just a common wisdom of cross-platform development, and not really specific to games or distributed systems.

Stronger than Platform-Independent: Strictly-Deterministic

The approach described above, is very well-known and is widely accepted as The Right Way to achieve platform-independence. However, having spent quite a bit of time with debugging of distributed systems, I've became a strong advocate of making your logic part not only platform-independent, but also *strictly-deterministic*. While strictly speaking, one is not a superset of the other one, in practice these two concepts are very closely interrelated.

The idea here is to have your game logic consisting of the functions, which are 100% defined by their-input-data plus by internal-game-logic-state; as we'll see below, strict determinism can be achieved when you call system-dependent functions inside, though it comes with some caveats and should be usually avoided. For most of the well-written code out there, a large part your game logic is already written more

"The idea here

or less around these lines, and there are only a few relatively minor modifications to be made. In fact, modifications can be that minor, that if your code is reasonably well-written and platform-independent, you may even be able to introduce strict determinism as an afterthought. I've done such things before, and it is not that much of a rocket science, but honestly, it is still much better to go for strict determinism from the very beginning, especially as the cost is very limited.

is to have your game logic consisting of the functions, which are 100% defined by their-input-data plus by internal-game-logic-state

Strictly-Deterministic Logic: Benefits

At this point, you should have two very reasonable questions. The first one is "What's in this strict determinism for me?", and the second one is "How to implement it?"

To answer the first question and to explain *why* you should undertake this effort, all the benefits of implementing your logic this way actually result from one single word: determinism. When all the outputs of your logic are completely defined by its internal state plus its inputs, your program module (class, etc.) becomes perfectly deterministic. And while you may think that this is a purely theoretical advantage, determinism provides several very practical benefits. Most of these benefits result from one all-important property of a strictly deterministic system:

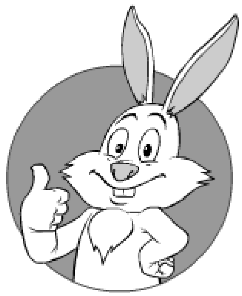
if you record all the inputs of a strictly deterministic system, and re-apply these inputs to another instance of the same strictly deterministic system in the same initial state, you will obtain exactly the same results

For practical purposes, let's assume that we have mechanics to write an *inputs-log*, recording all the inputs to our strictly-deteministic logic (see "Implementing Strictly-Deterministic Logic: Definitions" section below for implementation details):

- Your testing becomes deterministic, reproducible, and reversible
  - it means that as soon as you've got a failure, you can repeat the whole thing, and get the failure at exactly the same place in code. Such 100% reproducibility, in particular, allows things such as "let's stop our execution at 5 iterations *before* the failure."[2] If you have ever debugged a distributed program with a difficult-to-reproduce bug, you will understand that one this single item is worth all kinds of trouble.
  - in addition, your testing becomes more meaningful; without 100% determinism, any testing has a chance to fail depending on certain

conditions, and having your tests failing randomly from time to time is the best way I know to start ignoring such sporadic failures (which often indicate race-related and next-to-impossible-to-figure-out bugs). On the other hand, with 100% determinism, each and every test failure means that there is a bug in your code, that cannot be ignored and needs to be fixed (and can be fixed too, improving quality of your production code significantly)

- 100% reproducible bugs during post-mortem, both client-side and server-side
    - if you can log all the inputs to your logic in production (and quite often you can, at least on circular basis, see "EventProcessor Variations: Circular Buffers" section below for details), then after your logic has failed in production, you can "replay" this *inputs-log* on your functionally identical in-house system, and the bug will be reproduced at the very same point where it has originally happened. Even better, your in-house system needs be only functionally identical to production one (i.e. performance is a non-issue, and any PC will do); also you are not required to replay the whole system, you can replay only suspicious module instead. Moreover, during such *inputs-log* replay it is not necessary to run it on the same time scale as it was run in production; it can be run either faster (for example, if there were many delays, and delays can be ignored during replay), or slower (if your test rig is slower than the production one).



> **After your logic has failed in production, you can "replay" this *inputs-log* on your functionally identical in-house system, and the bug will be reproduced at the very same point where it has originally happened.**

- Regression testing using production data
    - if you've got your *inputs-log* just once, you can "replay" it to make sure that your code changes are still working. In practice, it comes handy in at least two all-important cases:
        - when your new code just adds new functionality, and unless this new functionality is activated, the system should behave exactly as before
        - when your new code is a pure optimization of previous one; when dealing with many thousands of simultaneous users, such optimizations can be Really Complicated (including major rewrites of certain pieces), and having an ability to make sure that new code works *exactly* as the old one (just faster), is extremely important.
- Keeping code bases in sync
    - If you're unlucky enough to have 2 code bases (or even "1.5 code bases", see Chapter [[TODO]] for details), then running the same *inputs-log* over

the two code bases provides an easy way to test whether the code bases are equivalent. Keep in mind that it requires cross-platform determinism, which has some additional issues, discussed in "Cross-Platform Issues" section below.

- User Replay, see discussion in "On User Replay" subsection below.
- Last but not least – determinism may allow you to run exactly the same logic (or even physics) both on client and server, feeding them with the same data and obtaining the same results. This will allow to save A LOT on network traffic (we'll discuss it in more detail in Chapter [[TODO]]); on the other hand, it requires cross-platform determinism across *all* your platforms, which is much more difficult to achieve than a single-platform one (and is more difficult to achieve than cross-platform determinism for two selected platforms).

Keeping in mind that:

- if you have a good development team, any reproducible bug is a dead bug
- the most elusive and by far time-consuming bugs in distributed systems tend to be race-related
- the race-related bugs are very difficult to reproduce, we can easily conclude that

having deterministic testing makes a Really Big Difference when it comes to distributed systems.

With strictly deterministic systems (and appropriate testing framework), all those elusive and next-to-impossible-to-locate race-related bugs are brought to you on a plate.[3]

There are also additional benefits of being deterministic[4], but these are beyond the scope of this book.

---

[2] to be fair, similar things in non-production environments are reportedly possible with GDB reverse debugging; however, it is platform-dependent, and is out of question for production, as running production code in reverse-enabled debug mode is tantamount to a suicide for performance reasons
[3] I don't want to say that you're like Pumba or Timon from Lion King series
[4] examples include, for example, an ability to perform incremental backup just by recording all the inputs (will work if you're careful enough), and an additional ability to apply an existing *inputs-log* to a recently fixed code base; the latter, while being quite esoteric, may even save your bacon in some cases, though admittedly rather exotic ones

Strictly-Deterministic Logic: On User Replay

With traffic being cheap and YouTube videos ubiquitous, User Replay is not that important these days, but still deserves being mentioned. When your game logic is fully deterministic, it will be possible for the player to record the game as it was played, get a very small (!) file with the record, and then share this file with the other players. Which in turn may help to build your community, etc. etc. As mentioned above, it is not *that* attractive these days, but you might still want to think about User Replay (which is coming to you more or less "for free", as you need determinism for other reasons too). If you add some interactive features during replay (such as changing viewing angle and commenting features such as labels attached to some important units, etc.), it might (or might not) have business sense.

A few points about implementing User Replay via deterministic replay:

- Usually, you need to record (and replay) only one entity – the one which is most close to the graphics. In other words, in terms of Generic QnFSM Architecture described below, you need to record/replay only "Animation&Rendering" FSM (and "Game Logic FSM" doesn't need to be recorded to enable User Replay)
- Keep in mind that User Replay will normally require you to adhere to the most stringent version of determinism, including cross-platform issues (see "Cross-Platform Issues" section below)
    - As a bit of relief, you MIGHT (or might not) be able to get away with not-that-strict determinism when it comes to floating-point issues (see "Cross-Platform Issues" section below); however, there is a big open question whether these really subtle differences will accumulate into some kind of macroscopic effects. [5]
- When implementing User Replay as deterministic replay, you'll need to deal with the "version curse". The problem here is that strictly speaking, replay won't run correctly on a different version of FSM 🙁 . So you will need to add FSM version number to all of these files, and then:
    - either to anayse track which version of replay file will run on which of the FSMs. I don't think is realistic (as analysis is too complicated)
    - or to keep *all* the different publicly-released versions of the FSM in the client, so *all of them* are available for replay. This one might fly, because FSM code size is usually fairly small (at most of the order of hundreds of kilobytes), and updates to post-Logic-to-Graphics Layer are relatively rare.
        - even in this case, your Animation&Rendering FSM will have external dependencies (such as DirectX/OpenGL), which can be updated

    and cause problems. However, as long as external dependencies are 100% backward-compatible – you should be fine (at least in theory)

    - while adding meshes/textures isn't a problem, replacing them is. For most of the purely cosmetic texture updates, you may be fine with using newer versions of textures on older replays, but for meshes/animations – probably not, so you may need to make them versioned too (ouch!)

- Ultimately, this is still a business decision, so even if you like the idea of User Replay a lot, but business guys say that they don't need this kind of stuff – don't bother with implementing it; while seemingly trivial, it will require quite a bit of time to implement in a way which makes it both replayable and interesting for your players.
    - On the other hand, decision whether you want to have determinism for testing/post-mortem purposes – is *not* a business decision, and you may (and IMNSHO should) do it pretty much regardless of Business Requirements (that is, unless Business Requirements state "crash for each user at least twice a day"); at the very least you should go for determinism for most of the games with Undefined Life Span.

---

[5] personally, I've never faced these things, so I cannot provide any real-world comments

Implementing Strictly-Deterministic Logic: Definitions

I hope I've managed to convince you that strictly-deterministic systems are a Good Thing™, and that now we can proceed to the second question: how to implement these strictly-deterministic systems?

"I hope I've managed to convince you that strictly-deterministic systems are a Good Thing™

First, let's define what we want from our strictly-deterministic system. Practically (and to get all those benefits above) we want to be able to run our code in one of three modes:

- **Normal Mode.** The system is just running, not actively using any of its strictly-deterministic properties
- **Recording Mode.** The system is running exactly as in Normal Mode, but is additionally producing *inputs-log*
- **Replay Mode.** The system is running using only information from *inputs-log* (and no other information), reproducing exact states and processing sequences which have occurred during Recording Mode

Note that Replay Mode doesn't require us to replay the whole system; in fact, we can replay only one part of the whole thing, the one which we suspect to be guilty. If after analysis we find that it was behaving correctly and that we have another suspect – we can replay that suspect from its own *inputs-log* (which hopefully has been written too during the same session which has caused failure).

Implementing Inputs-Log

Implementation-wise, *inputs-log* is usually organized as a sequence of "frames", with each "frame" depending on the type of data being written. Each of the "frames" usually consists of a type, and serialized data depending on type.

Let's discuss how it can/should be done in C++ (other languages are usually simpler, or MUCH simpler), and which caveats need to be avoided. Below are a few hints in this regard:

- don't serialize your data as plain C structures; use serialization library instead.
  - it is often a good idea to use your marshalling library (see Chapter [[TODO]]) for serialization purposes too
- it doesn't really make much difference which serialization format (binary or text-based) you use; it is much more important to encapsulate your serialization format from your state machine logic (i.e. to have serialization library which takes care of all the formatting stuff)
  - as a part of this encapsulation, I strongly suggest to define your own (and opaque!) stream class , using this your-own-and-opaque-class as a parameter to your serialization functions. The only thing which you're allowed to do with an object of this class, is to pass it to a function from your serialization library. This approach will save you quite a lot of trouble down the road.
  - despite exact format being more or less irrelevant, make sure that your serialization format is portable between your plaftorms

"it is much more important to encapsulate your serialization format from your state machine logic

- while we're speaking about *inputs-log,* most of the time your data will be plain and without any pointers; however, in some cases (and when state serialization becomes necessary, see, for example, "EventProcessor Variations: Circular Buffers" below), the need to serialize more complicated data structures may arise.
  - In such cases, usually, you fill find that your data (whether for *inputs-log* frames or for states) is still simple enough to be described by levels 1 to 3 on the sophistication scale as defined by [ISOCPP]. You may want to use

suggestions described there.

Implementing Strictly-Deterministic Logic: Original Non-Strictly-Deteministic Code

Now as we have our *inputs-log,* let's see how we can implement our logic which will record/replay it. Let's start with a simple example: a class, which implements a "double-hit" logic. The idea is that if the same NPC gets hit twice within certain pre-defined time, something nasty happens to him.[6] Usually, such a class would be implemented along the following lines:

```
class DoubleHit {
 private:
 const int THRESHOLD = 5;//in MyTimestamp units
 MyTimestamp last_hit;
  //actual type of MyTimestamp may vary
  // from time_t to microseconds, and is not important for our purposes

 public:
 DoubleHit() {
  last_hit = MYTIMESTAMP_MINUS_INFINITY;
 }

 void hit() {
  MyTimestamp now = my_get_current_time();
  if(now – last_hit < THRESHOLD)
   on_double_hit();

  last_hit = now;
 }

 void on_double_hit() {
  //do something nasty to the NPC
 }
};
```

While this example is intentionally trivial, it does illustrate the key point. Namely, while being trivial, function DoubleHit::hit() it is NOT strictly-deterministic. When we're calling hit(), the result depends not only on input parameters of hit() and on members of DoubleHit class, but also on the *time* when it was called.

[6] while such logic normally belongs to server-side in MMOs, it may need to be duplicated on the client-side (for example, due to the client-side prediction, see Chapter [[TODO]] for details), so it is relevant for client-side too

Implementing Strictly-Deterministic Logic: Strictly-Deteministic Code via Intercepting Calls

Now let's see what we can possibly do to make our DoubleHit::hit() deterministic. In general, I know two and a half approaches to achieve it.

> “The first approach is to “intercept” all the calls to the function my_get_current_time().

The first approach is to “intercept” all the calls to the function my_get_current_time(). “Intercepting calls” here is meant as changing behaviour of the function depending on the mode in which the code is running, adding/changing some functionality in “Recording” or “Replay” modes. “Intercepting” my_get_current_time() can be done, for example, as follows: whenever the system is running in “recording” mode, my_get_current_time() would run as normal, but would additionally store each returned value to the *inputs-log*. And whenever the system is running in “replay” mode, my_get_current_time() would read the next value from the *inputs-log*, and would return that value regardless of actual time (and without making any system calls). This is possible exactly because of 100% determinism: as all sequences of calls during Replay are exactly the same as they were during Recording, it means that whenever we're calling my_get_current_time(), at the “current position” within our *inputs-log* we will always have exactly a record which was made by my_get_current_time() during Recording.

Therefore, “interception” of the function my_get_current_time() may be implemented, for example, as follows:

```
MyTimestamp my_get_current_time() {
 if(Mode == REPLAY_MODE) {
  MyTimestamp ret =
    //read next frame from global inputs-log into 'ret'
    // this frame MUST be a my_get_current_time() frame
   ;
  return ret;
 }

 MyTimestamp ret =
   // code for my_get_current_time() before “call interception”
   ;

 if(Mode == RECORDING_MODE) {
   //write my_get_current_time() as a 'frame'
   //  to a global inputs-log
 }

 return ret;
}
```

Bingo! This approach would make our implementation strictly-deterministic, and without any code changes too! Actually, this is pretty much what [liblog] replay tool[7] did.

However, there is a significant caveat with this way of making your logic strictly deterministic. If we add (or remove) any calls to my_get_current_time() (or more generally, to any of the functions-which-record-to-inputs-log), the replay will fall apart. While replay will still work for exactly the same code base, things such as replay-based regression testing will become pretty much unusable in practice, and existing real-world *inputs-logs* (which are an important asset, helping to test things) will be invalidated too frequently.

---

[7] not to be confused with other tools with the same name; as of now, I wasn't able to find an available implementation of liblog 🙁

Implementing Strictly-Deterministic Logic: "Pure Logic"

An alternative way (the one which I usually prefer) of making the class strictly-deterministic, is to change the class DoubleHit itself so that it becomes strictly deterministic without any interception trickery. For example, we could change our DoubleHit::hit() function to the following:

```
void hit(MyTimestamp now) {
  if(now – last_hit < THRESHOLD)
    on_double_hit();
  last_hit = now;
}
```

If we change our class DoubleHit in this manner, it becomes strictly deterministic without any need to "intercept" any calls; let's name such classes "pure logic" classes.

In general, whenever there is a choice, I usually prefer this "Pure Logic" approach; it is more explicit than intercepting calls, more easily readable, and has better resilience to modifications. However, it has some implications to keep in mind:

- with "pure logic", it becomes a responsibility of the caller to provide stuff such as timestamps
  - this passing parameters may (and usually will) go through multiple levels of calls
  - at some level, however, some caller-of-caller-… needs to call my_get_current_time() and pass obtained value as parameter
- it is a responsibility of whoever-calls-my_get_current_time(), to record data to *inputs-log* (and to handle replay too). See class EventProcessor below for an example.

- the whole chunk of processing (while caller-which-has-called-my_get_current_time() passes parameter around) is deemed to happen at the same point in time. While this is exactly what is desired for 99.9% of game logic, you need to be careful not to miss remaining 0.1%.

Implementing Strictly-Deterministic Logic: TLS-based Compromise

As an alternative to passing parameters around, you might opt to pass parameters via TLS instead of stack.The idea is to store MyTimestamp (alongside with any other parameters of similar nature) to the TLS, and then whenever my_current_get_time() is called, merely read the value from TLS.

TLS

Thread-local storage (TLS) is a computer programming method that uses static or global memory local to a thread.

— *Wikipedia* —

In practice, it means doing the following:

- keep your original logic code intact, with my_get_current_time() calls within
- rename my_get_current_time() to my_real_get_current_time()
- at those points where you'd call my_get_current_time() (for passing result as a parameter) in "pure logic" model, call my_real_get_current_time() and write the result to TLS[8]
- implement my_get_current_time() as simply reading of the value from TLS

This model is a kind of compromise between the two approaches above; it is less verbose (and less explicit) then "pure logic", but it is functionally equivalent to "pure logic", and therefore it doesn't suffer when somebody inserts yet another my_get_current_time(). If you prefer this model to "Pure Logic" – it is fine, but you'll need to figure out fine details of TLS yourself, as I will describe things mostly in terms of "Pure Logic" (though it can be converted to TLS-based Compromise Model in a very straightforward manner).

In TLS-based Compromise Model, handling of the recording/replay is exactly the same as in "pure logic" model (see also class EventProcessor below); the only thing which TLS-based Compromise changes compared to the "pure logic", is how the data is passed from caller to callee; everything else (including the data written to the *inputs-log*) is exactly the same.

---

[8] for C++, see C++11's *thread_local* storage duration specifier, but there are usually other platform-dependent alternatives

Implementing Strictly-Deterministic Logic: Passing Input Parameters as Data Members

Yet another way to handle it is to put all the input parameters as data members of your class DoubleHit:

```
class DoubleHit {
 private:
 const int THRESHOLD = 5;
 MyTimestamp last_hit;
 MyTimestamp now;//NOT recommended because of confusion!

 public:
 //...
 void hit() {
  if(now – last_hit < THRESHOLD)
   on_double_hit();

  last_hit = now;
 }
 //...
```

While it again is functionally equivalent to "Pure Logic", and will work, I shall say that I don't like it on readability grounds. Perceptionally, members of class DoubleHit clearly represent "current state" of class DoubleHit, and putting *now* (which is an "input parameter", and is semantically very different from "current state") there will be too confusing as soon as you give the code to somebody not familiar with your conventions.

Implementing Strictly-Deterministic Logic: Which Model to Choose?

Personally, I usually prefer "Pure Logic" approach described above. However, I admit that "TLS-based Compromise" is functionally equivalent to "Pure Logic" one, and that a discussion about being explicit vs being brief in this case is pretty much about personal preferences.

Therefore, I think that any of these two models is fine, just stay away from "intercepting calls" and "passing parameters as data members"; actually, even "intercepting calls" and "passing parameters as data members" are light years ahead of having no strict determinism at all, but why settling for something worse when you can have something better at the same price?

Implementing Strictly-Deterministic Logic: Which system functions we're speaking about?

In general,

each and every of system calls (including system calls made indirectly via wrappers), creates a danger of deviating your class from being strictly-deterministic.

As a result, some of the readers will say: “hey, this way we will end up with millions of parameters (or function calls we need to intercept)!”. Fortunately, it is not that bad.[9] Let’s take a closer look at the question “what exactly do we need to intercept/provide?”

“Let's take a closer look at the question “what exactly do we need to intercept / provide?”

As we’ve already discussed in “Logic-to-Graphics Layer” section above, all the code which works directly with graphics, should be separated from game logic by Logic-to-Graphics Layer; moreover, the interface which we’ve defined for this Layer, was essentially one-way communication (with game logic sending instructions to draw something, to the Layer), and one-way communication which goes in “from logic” direction, doesn’t affect determinism. In a similar manner, all the code which directly calls network sockets or input, should be moved to the platform-dependent part; moreover, these parts will usually reside in a different thread, or at least “higher” than our game logic (so that will act as callers with regards to game logic), see section [[TODO]] below; such usage won’t affect determinism either.

This leaves us with two major items which are closely related to the logic, and are not deterministic per se; these are time and client-side configuration.[10] Let’s take a look at each of these two items:

- Time. Time as such is not deterministic by design, but obtaining time is generally cheap, so usually there is no problem for the caller to pass time to the callee, even if callee won’t use it. Therefore, for time we can use “pure logic” approach above, to make the things more explicit (and to avoid problems when existing input-logs get incorrect when somebody inserts another my_get_current_time() into processing logic); functionally equivalent “TLS Compromise” will be less explicit and less verbose too.
- Client-side Configuration. Client-side configuration is pretty much the only case when you may need an access to the client-side file system (leaving aside caching, see on it below). With regards to the client-side configuration, you usually can set it to one fixed value for the whole session, and to put this one fixed value into the very beginning of your input-logs. If you want to test *manipulating* client-side configuration (which I never needed and never heard of somebody who needed it, but in the game world anything can happen) – you may choose either to intercept calls, or to use a kind of exception-based

trickery which will be described in Chapter [[TODO]] with regards to conditional handling of real (hardware-based) randomicity.

- Other stuff. There is a chance that you will genuinely need to use a non-deterministic call which is not covered in this Chapter, and to do it from within your game logic. One such example includes re-formatting of the server time into local time (which is better to be avoided anyway to avoid confusion, replacing it with system-independent "it happens in 37 minutes from now" kind of stuff, but sometimes you just don't have a choice). In such cases, you have the same two choices as right above – either to intercept relevant calls[11], or to use exception-based approach described in Chapter [[TODO]]. As long as such calls are rare, both these approaches will work reasonably well in practice.

---

[9] it took me quite a few years to realize, how actually *good* it is
[10] for server-side, there is also real (hardware-based) randomicity and databases, but we'll set this discussion aside until Chapter [[TODO]]
[11] it is MUCH better to do this interception after conversion to system-independent data formats, so it is system-independent data which gets into *inputs-log*

Strictly-Deterministic Logic: Non-Issues

In addition to the non-deterministic issues described above, there are also three non-issues; these are pseudo-random numbers, logging and caching.

Pseudo-random numbers as such are perfectly deterministic; that is, as long as you're storing PRNG state as a part of your logical state (if you're using FiniteStateMachine as described below – as a member of specific class MyFiniteStateMachine). Instead of using non-deterministic rand() (which implicitly uses a global, see also below), you can either implement your own linear congruential PRNG (which is literally a one-liner, but is not really good when it comes to randomicity, see also Chapter [[TODO]]), or use one of those Mersenne Twister classes which are dime a dozen these days (just make sure that those PRNG classes have PRNG state as a class member, not as a global); for C++ you can use something like boost::mt19937. Note that to get your PRNG seeded, you still need to provide some seed which is external to your deterministic logic, but this is rarely a problem.

Logging/tracing (as in "log something to a text/binary log file"), while it does interact with an outside world, is usually strictly deterministic per se. Moreover, even if your logging procedure prints current time itself (and to do it, calls my_get_current_time() or something else of the sort), and technically becomes non-deterministic from the "all the world outside of our logic" point of view (this happens because it's end-result depends on the current time), it still stays strictly deterministic from the point of view of the logic itself (as long as the logic cannot read the log).[12] Practical consequence: even in “Pure Logic” model, there is no need to pass ‘now’ parameter to those framework-level functions which implement logging (even if they call my_get_current_time() inside, but only as long as the result of this my_get_current_time() is not used other than to write data to the log).

The second deterministic non-issue is related to caching. Caching (whether file-based or memory-based), when it comes to the determinism, is an interesting beast. As long as all your caching does, is strictly caching of the data and nothing else, it is deterministic, regardless of all the reads and writes (provided that original state of the cache is stored, if applicable, in *inputs-logs*). While relying on caching being implemented as a correct cache won’t allow you to “replay-test” caching itself, as long as you’re sure that your caching is working – you can rely on it’s determinism.[13]

“The second deterministic non-issue is related to caching.

More generally, such things as logging and caching (if they are strictly deterministic themselves), can be considered “outside” of our logic (more strictly – outside of “isolation perimeter” as defined in “Event-Driven Programming and Finite State Machines” section below); this approach greatly reduces amount of logging which is required to guarantee correct recording/replay, at the cost of the recording/replay being unable to aid with testing of your logging/caching routines. In practice, as logging/caching are relatively simple and are rarely changed, the latter restriction doesn’t cause too much trouble.

---

[12] I know that this explanation reads quite ugly, but I cannot find better wording now; regardless of the wording, the statements in this paragraph stand
[13] strict proof is beyond the scope of this book

Strictly-Deterministic Logic: No Access to Globals

This might go without saying, but let’s make it explicit:

for your logic to be strictly-deterministic, you MUST NOT use any global variables. Yes, it means “No Singletons” too.

Actually, it is not just a requirement to be strictly-deterministic, but is a well-known “best practice” for your code to be reasonably reliable and readable, so please don’t take it as an additional burden which you’re doing just to become strictly-deterministic; following this practice will make your code better in the medium- and long-run even if you're not using any of the benefits provided by strict determinism.

The only exception to this rule is that accessing constants is allowed without restrictions (well, as long they you don't modify them 😉).

As an consequence,

you SHOULD NOT use any function which implicitly uses globals

Identifying such functions can be not too trivial, but if you need to stay strictly deterministic, there it is a requirement to avoid them. Alternatively, you may decide to "intercept" these calls (and write whatever-they-return into *inputs-log*) to keep your logic strictly deterministic, but as noted above, "intercepting calls" is better to be avoided when feasible.

C standard library is particularly guilty of providing functions which implicitly access globals (this includes *rand()*). Most of these functions (such as strtok()) should be avoided anyway due to the logic being non-obvious and being potentially thread-unsafe on some of the platforms. One list of such functions can be found in [ARM]; note that the problem here is not about thread-safety, so rand() and strtok() are still non-deterministic even on those platforms (notably Windows) which make them thread-safe by replacing globals with TLS-based stuff.

Strictly-Deterministic Logic: Pointers

C/C++ pointers are quite a nasty beast in general, and can cause quite a few problems when it comes to determinism too ☹️ . The problem with pointers from determinism point of view is that in general, you cannot guarantee that allocated pointers are the same on different runs of the program. [14] As a result, below is a list of things which should be avoided when writing for determinism:

- using convoluted pointer arithmetic (and "convoluted" here means "anything beyond simple array indexing). Seriously, if you're relying on this kind of stuff, you'd better write for Obfuscated C contest and stay away from any serious development.
- sending pointers over the network (and writing them to *inputs-log*), regardless of marshalling used. Actually, this one should be avoided regardless of determinism.
- using pointers as identifiers
- using pointers for ordering purposes; even using pointers to get "just some

kind of temporary ordering” is not good for determinism, sorry about that

While this looks as quite a few items to remember about, it is not too bad in practice.

Strictly-Deterministic Logic: Cross-Platform Issues

Achieving strict determinism on one single platform is significantly easier than across different platforms. For many practical purposes (such as post-mortem and debugging), it is sufficient to have strict determinism only within one single platform. However, to obtain some other properties (for example, cross-platform-equivalence testing, User Replay, and identically running physics engines) you may need to have cross-platform determinism. In such a case, additional considerations apply:

- non-ordered and partially-ordered collections may produce different results on different platforms while staying compliant. For C++, examples include hash-table-based unordered_map<>/unordered_set<> containers, and tree-based partially ordered multiset<>/multimap<> containers.
    - a funny thing about them is that they ARE indeed nothing more than “moving bits around”, it is just that bits are moved in a bit different (but compliant) manner for different implementations
    - it means that one way to deal with them, is to write your own version (or just to compile *any* of existing ones to *all* the platforms); as long as the code for all the platforms is (substantially) the same, it will compile into the code which behaves exactly the same
    - for tree-based partially ordered sets/maps, you often can make them fully-ordered by adding an artificial ID (for example, incremented for each insert to the container) and using it as a tie-breaker if original comparison returns that objects are equal. It is quite a nasty hack, but if you don’t need to care about ID wraparounds (which is almost universally the case if you’re using 64-bit IDs), and you don’t care about storing an extra ID for each item in collection, it works.
- floating-point arithmetic issues. In short: while floating-point will return *almost* the same results on different platforms, making them *exactly* the same across different hardware/compilers/… is very challenging at the very least. For further details, refer to [RandomASCII2013] and [Fiedler2015]. A few minor but important points *in addition to the discussion in those references*:
    - As floating-point arithmetic is once again all about “moving bits around” (it just takes some bunches of bits and returns some other bunches of bits), it can be made perfectly deterministic. In practice, you can achieve it by using software floating-point library which simulates floating-point via integer arithmetic [[TODO: add ref to Knuth]]
        - Note that such a library (if used consistently for *all* your platforms)

does *not* need to be IEEE compliant; all you need is just to get some reasonable results, and last bit of mantissa really matters in practice (as long as it is the same for all the platforms)

- such libraries are slooooow; for a reasonably good floating-point emulation library (such as [SoftFloat]) you can expect slowdown of the order of 20-50x compared to hardware floating point 🙁 .
  - however, certain speed up can be expected if the library is rewritten to avoid packing/unpacking floats (i.e that class MyFloat is actually a two-field struct), and replacing IEEE-compliant rounding with some-reasonable-and-convenient-to-implement rounding; very wild guesstimate for such an improvement is of the order of 2x [JohnHauser], which is not bad, but will still leave us at least with 10x slow-down compared to hardware floating point 🙁 .
- however, if you're fine with this 20-50x-slower floating-point arithmetic (for example, because your logic performs relatively few floating-point operations) – such libraries will provide you with perfect determinism.

Strictly-Deterministic Logic: Implementation summary

Given analysis above, we've found that while there are tons of places where your logic can potentially call external functions (and get something from them, making the logic potentially non-deterministic), in practice all of them can be dealt with easily, and in most cases the only thing you'll need to pass around, will be "current timestamp".

All the other system function calls will fall under one of the following:

- function calls which are output-only (drawing, logging, generating output events)
- function calls which shouldn't be called from within the logic (communications, user input)
- function calls which are used to implement caching; as long as caching is working correctly, they can be ignored for the purposes of determinism (see explanation above)

Even if you need more than "current timestamp", nothing prevents you from making a struct, consisting of all-of-your-pre-calculated-input-parameters, and passing around one single parameter (*FSMInputData* input_data* or something).

From my experience, this single extra parameter is not a large price for all the benefits you will get from making your logic strictly deterministic (and if you have strong feelings about this extra parameter, you can avoid it by using TLS Compromise).

> "This single extra parameter is not a large price for all the benefits you will get from making your logic strictly deterministic (and if you have strong feelings about this extra parameter, you can avoid it by using TLS Compromise)

As for other issues (those not related to external function calls), they are also of only very limited nature until you're going for a full-scale cross-platform determinism; neither avoiding globals (which is a good practice anyway), nor avoiding pointer-related trickery tends to cause much practical problems.

However, if you're going into realm of cross-platform determinism, things may get quite a bit nastier (and will cause more trouble); while collection differences can be handled if you're careful enough, achieving fully cross-platform floating point calculations can be trouble across different CPUs.

Strictly-Deterministic Logic: Overall summary

TL;DR of the "Stronger than Platform-Independent: Strictly-Deterministic" section:

- Strictly-deterministic logic is a Good Thing™, providing game-changing benefits for debugging distributed systems, including production post-mortem
- When implementing strictly-deterministic logic, either "Pure Logic" approach, or "TLS-based Compromise" is generally preferred
- Implementing strictly-deterministic logic requires rather few changes in addition to following existing best practices, as long as cross-platform determinism is not required
- Going for a full-scale cross-platform determinism can be tricky, especially because of floating-point issues.

Event-Driven Programming and Finite State Machines

when they don't know what to say
and have completely given up on the play
just like a finger they lift the machine
and the spectators are satisfied
— Antiphanes, IV century B.C. —

So, we've got our one single DoubleHit class as a strictly-deterministic logic. Good for us, but in any realistic system there will be much more classes than this. What should we do about it?

Pieces of strictly-deterministic logic can be combined with each other easily; the only two things to keep in mind are the following:

- don't mix strictly-deterministic code with any non-strictly-deterministic code; such a mix will be non-strictly-deterministic, and you will lose all the benefits arising from being strictly-deterministic
- if you're using "Pure Logic" model to achieve determinism, you're not allowed to call my_get_current_timestamp() within your logic. It implies that whenever you need to pass the MyTimestamp parameter to the callee, you yourself should get it from the caller.

The latter observation leads us to a reasonable question: well, somebody will need to call my_get_current_timestamp(), so where this whole calling tree (the one which passes 'now' around) should end? Let's see how it can (and should) be organized.

> **"We need to make an "isolation perimeter" where we control and log all the inputs of this piece of code.**

First of all, let's note that to take advantage of determinism of a certain piece of code, we need to isolate it and make sure that we can control (and log to inputs-log) *all* the inputs of this piece of code. In other words, we need to make an "isolation perimeter" where we control and log all the inputs of this piece of code. Now let's see how we want to build this "isolation perimeter". Systems such as [liblog] are trying to build this "isolation perimeter" around the whole app; actually, without access to internals of the code it is next to impossible for them to do anything else. On the other hand, we do have access to internals of our own code, and we can build our "isolation perimeter" pretty much anywhere. Let's discuss one approach which has been observed to produce very good results in practice.

Let's say that we have a high-level class EventProcessor, which does nothing but processes incoming events (anything can be an event, from the user input up to passing a certain amount of time, with message-from-server in between). At the point of receiving the event, EventProcessor calls my_get_current_time(), and then calls FiniteStateMachineBase::process_event():[15]

```
class FiniteStateMachineBase {
 public:
 virtual void process_event(MyTimestamp now, MyEvent& ev) = 0;
};

class EventProcessor {
 private:
 FiniteStateMachineBase* fsm;
 int mode;
 InputsLogForWriting& ol;

 public:
 void process_event(MyEvent& ev) {
  MyTimestamp now = my_get_current_timestamp();
  if(mode == RECORDING_MODE) {
   //write both ev and now to ol
  }
  try {
   fsm->process_event(now,ev);
  } catch(exception& x) {
   //some error handling
  }
 }

 void replay_event(InputsLogForReading& il) {
  //parse inputs-log and call fsm->process_event() accordingly
 }
};
```

In “Recording” mode, EventProcessor::process_event() will additionally write both *now* and *ev* to inputs-log. In addition, while strictly not required to ensure determinism, usually at least some of the output-only function calls (such as events-generated-for-other-FSMs, instructions to Logic-to-Graphics Layer, etc.) are also written to the same inputs-log, to simplify automated testing and debugging.

In “Replay” Mode, EventProcessor::process_event() is not called at all; instead, EventProcessor::replay_event() is called, which parses an entry in the inputs-log and calls fsm->process_event() accordingly; in addition, EventProcessor::replay_event() may parse expected outputs from the inputs-log and compare them with the calls which actually happened, raising an exception at the first sign of inconsistency.

Both class FiniteStateMachineBase and class EventProcessor mentioned above are merely providing a framework to implement your own FiniteStateMachine to implement some kind of specific logic:

```
class MyFiniteStateMachine : public class FiniteStateMachineBase {
  private:
  //FSM state goes here
  public:
  virtual void process_event(MyTimestamp now, MyEvent& ev) override {
    //within, the function MAY generate output,
    // including sending events intended for other state machines (!)
    // ...
  }
};
```

It is these classes-derived-from-FiniteStateMachineBase which contains actual FSM state (as data members) and actual FSM logic (as process_event() function).

Relation to Finite Automata as taught in Uni

> *– Have it compose a poem – a poem about a haircut! But lofty, noble, tragic, timeless, full of love, treachery, retribution, quiet heroism in the face of certain doom! Six lines, cleverly rhymed, and every word beginning with the letter s!! - And why not throw in a full exposition of the general theory of nonlinear automata while you're at it?*
> *— Dialogue between Klapaucius and Trurl from The Cyberiad by Stanislaw Lem —*

NB: if you're not interested in theory, you can safely skip this subsection; for practical purposes it suffices to know that whatever event-driven program you've already written, is in fact a finite automaton, so there is absolutely no need to be scared. On the other hand, if you ***are*** *interested in theory, you'll certainly need much more than this subsection. The idea here is just to provide some kind of "bridge" between your uni courses and practical use of finite automata in programming (which unfortunately differ significantly from quite a few courses out there).*

First of all we need to note that our class FiniteStateMachine, strictly falls under definition of Finite Automaton (or more precisely – Deterministic Finite Automaton) given in Wikipedia (and in quite a few uni courses). Namely, deterministic Finite State Machine (a.k.a. Deterministic Finite Automaton) is usually defined as follows [Wiki.DeterministicFiniteAutomaton]:

- Σ is the input alphabet (a finite, non-empty set of symbols).
  - in our FiniteStateMachine, Σ is a set of values which a pair (now,ev) can take; while this set is exponentially huge, it is still obviously finite
- S is a finite, non-empty set of states.
  - in our case, it is represented by all possible combinations of all the bits forming data members of FiniteStateMachine. Again, it is exponentially huge, but certainly still finite.
- s_0 is an initial state, an element of S.
  - whatever state results from FiniteStateMachine::FiniteStateMachine()

- δ is the state-transition function. $\delta: S \times \Sigma \rightarrow S$
  - implemented as FiniteStateMachine::process_event();
- F is the set of final states, a (possibly empty) subset of S.
  - for our FiniteStateMachine, F is always empty.

Therefore, our class FiniteStateMachine complies with this defintion, and *is* a Deterministic Finite Automaton (and most of event-processing systems are Finite Automatons, albeit usually non-deterministic ones).

Quite often,[16] in university courses state-transition function δ is replaced with a "set of transitions". From formal point of view, these two definitions are strictly equivalent, because:

- for any state-transition function δ with a finite number of possible inputs, we can run this function through all the possible inputs, and to obtain the equivalent set of transitions.[17]
- having a set of transitions, we can easily define our state-transition function δ via this set

"The problem which kills this neat idea, is known as "state explosion", and is all about exponential growth of states as you increase complexity of your machine.

On the other hand, if you start to define your state machine via set of transitions *in practice (and not just in theory),* most likely you're starting a journey on a long and painful way on shooting yourself in the foot. When used in practice, this "set of transitions" is usually implemented as some kind of a state transition table (see [Wiki.StateTransitionTable]). It all looks very neat, and fairly obvious. There is only one problem with table-driven finite state machines, and the problem is that they don't work in the real world. The problem which kills this neat idea, is known as "state explosion", and is all about exponential growth of states as you increase complexity of your machine. I won't delve into too much details on the "state explosion", but will note that it quickly becomes really really bad as soon as you're starting to develop something realistic; even having 5 different 1-bit fields within your state lead to a state transition table of size 32, and adding anything else is already difficult; make it 8 1-bit fields and corresponding 256 already existing transitions, and adding any further logic has already became unmanageable.[18] In fact, while I've seen several attempts to define state machines via state transition tables, none of them was able to come even somewhat-close to succeeding.

What is normally used in practice, is an automaton which is defined via state-transition function δ (which function δ is implemented as a usual function in an imperative programming language, see, for example, FiniteStateMachine::process_event() above). Actually, such automatons are used much more frequently than developers realize that they're writing a finite automaton 😉 . To distinguish these real-world state machines from table-driven (but usually impractical) finite state machines, I like the term "ad-hoc state machines" (to the best of my knowledge, the term was coined in [Calderone2013]).

---

14 for some of the platforms, *and* when you have your whole program recorded/replayed, you may get such guarantees, but as we're aiming to record/replay on smaller-than-whole-program basis, it won't fly for us
15 yes, it could have be done with less code, but I certainly prefer to be extremely explicit here
16 see, for example, [CSC173]
17 Never mind that such enumeration may easily take much longer than the universe ends from something such as Heat Death or Big Rip – in maths world we don't need to restrict ourselves with such silly notions.
18 while hierarchical state machines may mitigate this problem a bit, in practice they become too intertwined if you're trying to keep your state machines small enough to be table-driven. In other words: while hierarchical state machines are a good idea in general, even they won't be able to allow you to use table-driven stuff

Implementing Deterministic Finite State Machines

Back to the real world, we need to discuss how to implement deterministic finite state machines. In general, while you're staying within your FiniteStateMachineBase interface and restrictions stated above, exact implementation is up to you, and may easily be different for the different state machines you have. Popular possibilities include:

"You can implement your Finite State Machine as a deterministic variation of a usual event-driven program

- any deterministic event-driven program (which is inherently a deterministic ad-hoc state machine); this is probably what you really want to do if it is your first experience with the state machines. While it may (or may not) result in the code which is unwieldy, it is a very familiar pattern, and (if you're making it deterministic as discussed above), you will still benefit from all the goodies mentioned in "Strictly-Deterministic Logic: Benefits" section (such as greatly improved debug and post-mortem).
- in many cases, it is useful to have a separate data member called *state,* which takes one of (mutually exclusive) enumerated values. One good example for *state* variable is your PC *running*, or *walking*, or *jumping*, or *croaching*; another good example is *state* reflecting stage of the specific quest. In any case, you're allowed to have other

data members in addition to *state* (they represent so-called "extended state" in terms of UML state machines)

- note that in quite a few cases, having *state* member is considered a requirement to be named a "Finite State Machine"; I hate arguing about which terminology is "right", so I will just note that we're using the definition of "Finite State Machine" taken from Wikipedia (see also above), and according to that definition, *state* member is not strictly required (though often convenient).
- Finite State Machines with *state* member can be implemented in an ad-hoc manner (basically with a *switch* on your *state* in quite a few places); this is simple and is known to work
- alternatively, you may want to use State pattern from [GameProgrammingPatterns.StatePattern]; the same book also gives some hints on implementing hierarchical state machines and push-down automata.
- If you have this *state* member, you may want to document a diagram of your state machine using UML state diagrams [Wiki.UMLStateMachine]; they have some useful concepts too. note that I mean using it only for documentation purposes (and not for code generation), so it doesn't really what kind of software you're using for drawing[19]

- As a Big Fat rule of thumb, you SHOULD NOT try table-driven state machines (those which you might have been taught in uni); see "Relation to Finite Automata as taught in Uni" section above if you need justification.

---

[19] for most of the commercial games, you will have a requirement to keep such things private, so double-check your policy before using something like draw.io; something like Visio will usually be ok

EventProcessor Variations: Circular Buffers

The implementation of EventProcessor described above, is certainly not the only possible one. In fact, the beauty (and practical implications) of the separation between EventProcessor and FiniteStateMachine is that we have a liberty to plug our FiniteStateMachine into pretty much any EventProcessor we want.

One practical case for a different EventProcessor arises when we want to have a "post-mortem log" (sufficient to identify the problem), but we don't want to write all the things "forever and ever", as it might cause performance degradation.

Ok, for such cases we can make a different EventProcessor, let's name it EventProcessorWithCircularBuffer. For this EventProcessorWithCircularBuffer, we can implement *inputs-log* as an in-memory circular buffer, avoiding the need to keep the data forever-and-ever. However, for this to work, it will additionally need to:

"We can implement *inputs-log* as an in-memory circular buffer, avoiding the need to keep the data forever-and-ever.

- make sure that underlying FiniteStateMachine has an additional function such as void serializeStateToLog(InputsLogForWriting& ol), and a counterpart function deserializeStateFromLog(InputsLogForReading& il). State serialization should be implemented in a manner which is consistent with serialization used for *inputs-log* in general; see "Implementing Inputs-Log" section above for further discussion on state serialization.
- call this serializeStateToLog() function so that in-memory circular buffer always has at least one instance of serialized state
- make sure that there is always a way to find serialized state even after a circular buffer wrap-around (this can be done by designing format of your inputs.log carefully)
- on failure, just dump the whole in-memory *inputs.log* to disk
- on start of "Replay", find the serialized state in inputs-log, call deserializeStateFromLog() from that serialized state, and proceed with rollforward as usual.

EventProcessorWithCircularBuffer describes only one of multiple possible implementations of EventProcessor; it has an advantage that all the logging can be kept in-memory and therefore very cheap, but in case of trouble this in-memory log can be dumped, usually providing sufficient information about those all-important "last seconds before the crash". Further implementation details (such as "whether implement buffer as a memory-mapped file" and/or "whether the buffer should be kept in a separate process to make the buffer corruption less likely in case of memory corruption in the process being logged") are entirely up to you 🙂 .

One very important usage of EventProcessorWithCircularBuffer is that in many cases it allows to keep the logging running all the time in production, both on client side and on server side. It means near-perfect post-mortem analysis in case of problems

Let's make some very rough estimates. Typical game client receives around a few hundred bytes per second; let's take it at 200 bytes/sec. User input is very rarely more than that. It means that we're speaking about at most 500-1000 bytes/second. 1MByte RAM buffer is nothing for client-side these days, and at a rate of 1000 bytes/second we'll be able to store about 3 hours of "last breath data" for our "game logic" FSM; these 3 hours of data is usually orders of magnitude more than enough to find a logical bug. For an FSM implementing your animation/rendering engine, calculations will be different, but taking into account that all the game resources are well-known and don't need to be recorded, we again can keep the data recorded to the minimum, again enabling a very good post-mortem.

For the server side, you will need much more memory to run recording, and you might not be able to keep circular buffers running all the time, but at the very least you should be able to run them on selected FSMs (those are currently under suspicion, or those which are not-so-time-critical, or just a random sample).

Deterministic Finite State Machines: Nothing too New But…

While there is nothing really new in event-driven programming (and ad-hoc finite state machines used for this purpose), our finite state machines have one significant advantage compared to those usually used in the industry. Our state machines are strictly-deterministic (at least when it comes to one single platform), that allows for lots of improvements for debugging of distributed systems (mostly due to "replay testing/debugging" and "production post-mortem").

On the other hand, in academy Deterministic Finite Automata are well-known, but I don't know of the descriptions on "how to write them in practical applications".

On the third hand 😉 , determinism for games has been a popular topic for a while (see, for example, [Gamasutra2001]) , and in recent years has got a new life with MMOs and synchronous physics simulation on client and server (see, for example, [Fiedler2015-2]).

On the fourth hand (yes, we're exactly half way on becoming an octopus), I didn't see anybody concentrating on using determinism for the purposes of debug and production post-mortem, and from my experience effect of these items on the quality of your game cannot be underestimated. If you want to have your game crashing 10x less frequently than competition – do yourself and your players a favour, and record production *inputs-logs* for post-mortem purposes, as well as perform replay-based testing. I know I sound like a commercial, but as a gamer myself I do have a very legitimate interest in making games crash much more rarely than they do it now; I also know that for most of good game developers out there, deterministic testing and post-mortem will help in this regard, and will help a lot (in addition to any replay/synchronous-physics goodies if you need them).

Deterministic Finite State Machines: Summary

To summarize all this long talk about determinism and state machines:

- for distributed systems, you DO need to have *at least* single-platform determinism. It will help A LOT with testing, debugging, and production post-mortem.
  - achieving this one is not too difficult, and is usually only a minor annoyance
  - on the other hand, it still provides A LOT of useful stuff, mostly for the purposes of bugfixing (including those elusive production-only bugs)
- for other purposes (cross-platform code equivalence testing, User Replay, and physics equivalence) you MAY need cross-platform determinism
  - achieving this one can be a challenge, especially in the field of floating-point calculations
- nothing prevents you from starting small, with single-platform determinism, and then trying to extend it to cross-platform one. Unless the life of your game depends on a cross-platform determinism, this might be a viable option to pursue.
- Finite State Machines are a nice and convenient way to express deterministic (sub)systems
  - this includes (but is not limited to) ad-hoc Finite State Machines, which are nothing more than very-well known event-based systems

[[To Be Continued…

This concludes beta Chapter V(c) from the upcoming book "Development and Deployment of Massively Multiplayer Games (from social games to MMOFPS, with social games in between)". Stay tuned for beta Chapter V(d), "Modular Architecture: Client-Side Overall Architecture.]]

[–] References

[ISOCPP] Standard C++ Foundation, "How do I select the best serialization technique?"
[liblog] Dennis Geels, Gautam Altekar, Scott Shenker, Ion Stoica, "Replay Debugging for Distributed Applications"
[ARM] "C library functions that are not thread-safe", ARM Compiler toolchain ARM C and C++ Libraries and Floating-Point Support Reference
[RandomASCII2014] Bruce Dawson, "Floating-Point Determinism", 2013
[Fiedler2015] Glenn Fiedler, "Floating Point Determinism", Gaffer on Games, 2015
[SoftFloat] "Berkeley SoftFloat"
[JohnHauser] John Hauser, author of Berkeley SoftFloat, "Private communications

with"
[Wiki.DeterministicFiniteAutomaton] Wikipedia, "Deterministic Finite Automaton"
[CSC173] Randal Nelson, "CSC 173: Computation and Formal Systems", University of Rochester
[Wiki.StateTransitionTable] Wikipedia, "State Transition Table"
[Calderone2013] Jean-Paul Calderone, "What is a State Machine?"
[GameProgrammingPatterns.StatePattern] Robert Nystrom, "Game Programming Patterns"
[Wiki.UMLStateMachine] Wikipedia, "UML State Machine"
[Gamasutra2001] Patrick Dickinson, "Instant Replay: Building a Game Engine with Reproducible Behavior", Gamasutra, 2001
[Fiedler2015-2] Glenn Fiedler, "Deterministic Lockstep", Gaffer on Games, 2015

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« Chapter V(b). Modular Architecture: Client-Side. Programmin...

Chapter V(d). Modular Architecture: Client-Side. Client Arch... »

Filed Under: Distributed Systems, Programming, System Architecture
Tagged With: client, debug, game, multi-player