Slides Are Available at http://ithare.com/

by 'No Bugs' Hare, nobugs@ithare.com

Deterministic Components for Interactive Distributed Systems

Benefits and Implementation

Part I. Determinism. Definitions and Benefits

Definition 1

non_deterministic == not_fully_testable

Definition 2

recording/replay

Same-Executable Determinism

vs Cross-Platform one

Benefits:

Testability

Replay-based regression testing

Equivalence testing, Fuzz Testing

Production post-factum debugging

Low-latency fault tolerance

Some Others

Part II. Implementing Deterministic Components

- *Isolation Perimeter*
- *Sources of non-Determinism*
- *Dealing with non-Determinism*
  - ***Multithreading***
    - (Re)Actors
    - Circular logging
  - System calls
    - ***Call wrapping***
    - Pre-Calculation
    - Non-Blocking Calls
  - Risky Behaviours
  - ***Compatibility Issues***
    - CPU, Compiler, Libraries
    - Floating-point Determinism
    - C++ vs Others
    - Don't Apply to Same-Executable Determinism
- ***Non-issues (PRNG, logging, caches)***

Part III. Building Interactive Distributed Systems

- *Properties*
- *Typical Structure*
- *The Problem*
- *The Solution*
- *Making System Deterministic as a Whole*

'NO BUGS' HARE

Slides Online

http://ithare.com/

Part I. Determinism. Definitions and Benefits

Definition 1:

A program is deterministic
if and only if
its outputs are 100% defined
by its inputs

Observation 1:
Non-deterministic program cannot be fully testable using only deterministic testing

Observation 2:
"Non-deterministic tests have two problems, firstly they are useless, secondly they are a virulent infection that can completely ruin your entire test suite." – Martin Fowler

Observation 3:
Non-deterministic programs are not fully testable

Defining Determinism, Take 1

Deterministic Example:

```
void f1(int a, int b) {
  printf(“%d\n”, a+b );
}
```

Fully Testable ✔

Non-Deterministic Example:

```
void f2(int a, int b) {
  time_t now = time(NULL);
  printf(
    “As of %s, a+b=%d\n”,
    asctime(localtime(&now)),
    a+b );
}
```

Not Fully Testable ✘

Definition 2:

For a program to be deterministic
it is sufficient that
(a) we can record all its inputs;
(b) when replaying these recorded inputs against the same program, we always get the same outputs

Definition 2a:

*Program is "**same-executable-deterministic**" if replay guarantees to produce the same result only when it is run on exactly the same executable as the one where recording was made*

Doable ✔

Definition 2b:

*Program is "**cross-platform-deterministic**" if replay guarantees to produce the same result on ANY platform as long as source code is the same*

Very Difficult ✘

100% Reproducibility

Reproducible Bug is a Dead Bug

Same-executable determinism is sufficient

Replay-based Regression Testing

Needs EXACTLY the same functionality to work

Solution:

- *split all changes intended for version N+1 into 2 categories:*
  - *not supposed to modify existing logic (this will include most of new functionality)*
  - *modifying existing logic*
- *make version N½ consisting only of version N + non-modifying changes, and replay-test it using records from version N.*

Required: Same-platform determinism against minor changes

Replay-Based Equivalence Testing

if you need to:
- test new implementation of the same thing, or
- separate code bases, or
- test equivalence under different platforms/compilers.

Fuzz Testing

- strictly speaking, fuzz testing does require determinism (but in practice does work without it <wink />)
- replayable records are an ideal substrate for fuzz testing
- fuzz tester such as afl will just mutate the records and replay them

Required determinism: depends

Production post-factum debugging

Ultimate developer's nightmare:
bug in production.

Holy grail of production debugging:
fix bugs from the very first occurrence
– ideally - reproduce it under debugger

With deterministic replay, it becomes perfectly possible. *Just record all inputs on the production box - and send them to developers after the problem occurs.*

Required: Same-executable determinism

Determinism Benefits - Production Debugging

Fragment from David Aldridge's presentation
"I Shot You First: Networking the Gameplay of HALO: REACH"
Courtesy of David Aldridge and GDC Vault

Low-Latency Fault Tolerance for Stateful Objects

Using determinism - it is possible to achieve low-latency fault tolerance. Very shortly:

- *we're recording inputs all the time (with record including state snapshots)*
- *record and main object are kept on different physical boxes*
- *in case of failure - object can be reconstructed from record-with-snapshot*
- *similar to "Virtual Lockstep"*

Low-Latency Migration of Stateful Objects

- *implementation is along the same lines*

Required determinism: Same-Executable

Determinism Benefits - Others

Deterministic Lockstep Protocol
Used in games and simulations.

User Replay
Used in games.

Determinism Required: Cross-Platform

Part II. Implementing Deterministic Components

Observation 4.

Program becomes deterministic as soon as we have eliminated all the sources of non-determinism

Observation 5.

As soon as we establish an "Isolation Perimeter" with everything inside the perimeter being deterministic, and recording all the data crossing the perimeter in the "inside" direction - the part of the Program within the Isolation Perimeter complies with our Definition 2.

Multithreading

System Calls
- *most of system calls are non-deterministic*
- *relief: we can try to exclude malloc() - though see below*

Risky Behaviours
- *non-initialised memory (more generally - relying on an Undefined Behaviour)*
- *relying on pointer values (incl. sorting)*

Compatibility Issues
- *CPU*
- *Compiler*
- *Libraries*

Multithreading

Enemy #1 of determinism is multi-threading. With multi-threading - you should consider your program non-deterministic until proven otherwise

This is related to an observation that timings in different threads are not guaranteed (at least because of external interrupts).

My Favourite Way to Deal with MT: (Re)Actors

- *a.k.a. Actors, Reactors, ad-hoc FSMs, and Event-Driven Programs*
- *very straightforward, and tend to perform very well*
- *contrary to popular belief - (Re)Actors are scalable too*
- *don't introduce non-determinism*
- *also it is very straightforward to record all the input events.*

There **are** *other architectures which allow to deal with multithreading in deterministic manner - but you'll need to prove correctness of them yourself.*

Generic (Re)Actor

```
class GenericReactor {
  virtual void react(const Event& ev) = 0;
};
```

Infrastructure Code - Event Loop

```
GenericReactor* r =
  reactorFactory.createReactor(...);
while(true) { //event loop
  Event ev = get_event();
    //from select(), libuv, ...
  r->react(ev);
}
```

Specific (Re)Actor

```
class SpecificReactor :public GenericReactor {
  void react(const Event& ev) override;
};
```

Recording Loop

```
while(true) {
  Event ev = get_event();
  if(mode == Recording)
    write_ev_log_frame(ev);
  r->react(ev);
}
```

Replaying Loop

```
while(true) {
  Event ev = read_ev_log_frame();
  r->react(ev);
}
```

Circular Inputs-Log

- No need to store ALL events from the very beginning
- Need to ensure that there is a serialised state within the inputs-log at all times
  - if necessary - we can try incremental serialization
- Can be in-memory one, to use only in case of problems

Sources of Non-Determinism

Multithreading

- (Re)Actors
- Circular Logging

System Calls

Risky Behaviours

Compatibility Issues

System Calls

- *As noted above, most of system calls are non-deterministic, including:*
  - *I/O*
  - *time etc.*
  - *real RNG*
  - *and so on*
- *However, I suggest to exclude malloc() etc. - and say that we do not rely on specific pointer values instead*

Non-deterministic example:

```
void f2(int a, int b) {
  time_t now = time(NULL); //(TROUBLE)
  printf(
    "As of %s, a+b=%d\n",
    asctime(localtime(&now)),
    a+b );
}
```

Let's deal with:

```
time_t now = time(NULL);
```

System Calls and Determinism: Call Wrapping

Non-deterministic:

```
time_t now = time(NULL);
```

Replace with deterministic:

```
time_t now = my_time();
```

Where:

```
time_t my_time() {
  if(mode==Recording) {
    time_t ret = time(NULL);
    write_time_log_frame(ret);
    return ret;
  }
  else {
    assert(mode==Replay);
    return read_time_log_frame();
  }
}
```

The Trick

*Due to deterministic nature of our program, all the calls will happen in **exactly** the same places in relation to input events and other calls, so whenever my_time() is called during replay - there will be a corresponding inputs-log frame waiting for us at the current position within the inputs-log.*

Formally - position of the my_time() frame within the inputs-log is a function of the previous inputs and return values of the previous calls, and as long as this function is deterministic - position is deterministic too.

Call Wrapping: Pros and Cons

Pros:

- works for ALL the system calls

– exceptions are related to returned pointers but are quite rare.

Cons:

- not resilient to small changes

– not a problem for Same-Executable Determinism, but is quite a headache for Equivalence Testing and Replay-Based Regression Testing

Version 1:

```
time_t t = my_time(NULL);
printf(“%d\n”, t);
//...
time_t t2 = my_time(NULL);
printf(“%d\n”, t2);
```

Version 2:

```
time_t t = my_time(NULL);
printf(“%d\n”, t);
//...
printf(“%d\n”, t);
```

System Calls and Determinism: Pre-Calculation

Field of Event:

```
time_t t = ev.current_time;
printf(“%d\n”, t);
//...
time_t t2 = ev.current_time;
printf(“%d\n”, t2);
```

TLS-based my_time2():

```
thread_local current_time;
    //pre-populated by Infrastructure Code
    //  before calling react()

time_t my_time2() {
    return current_time;
}
```

Blocking version:

```
switch( ev.type ) {
  case EVENT_A: {
    do_something1();
    X x = long_call();
    do_something2();
  } break;
}
```

Non-Blocking version:

```
switch( ev.type ) {
  case EVENT_A:
    do_something1();
    start_long_call();
   break;
  case LONG_CALL_RETURNED: {
    X x = ev.parse_return();
    do_something2();
  } break;
}
```

Sources of Non-Determinism

Multithreading

- (Re)Actors
- Circular Logging

System Calls

- Call Wrapping
- Pre-Calculation
- Non-Blocking Calls

Risky Behaviours

Compatibility Issues

Risky Behaviours

- *Undefined Behaviours:*
  - *reading uninitialized memory*
  - *violating strict weak ordering for STL containers*
  - *etc.*

- *Using Unsupported Inter-Thread Communication Mechanisms.*
  - *No non-const globals(!)*

- *Relying on pointer values*
  - *we MUST NOT do ANYTHING but dereferencing*
  - *Can be avoided entirely if we "wrap" malloc() and guarantee stack location, but is usually too expensive this way.*

Sources of Non-Determinism

Multithreading

- (Re)Actors
- Circular Logging

System Calls

- Call Wrapping
- Pre-Calculation
- Non-Blocking Calls

Risky Behaviours

- Under our Control
- Feasible to Avoid

Compatibility Issues

Compatibility Issues

- *Sources:*
  - *CPU*
  - *compiler (and compiler settings)*
  - *libraries*

Compatibility Issues

- *Special Case: Floating-point Determinism*
  - *Particularly Nasty, especially for C/C++*
  - *Non-associative: (a+b)+c != a+(b+c)*
  - *Library functions (sin() etc.)*

Compatibility Issues

- ***C/C++***: *pretty bad*
  - *LOTS of UB*
  - *floating point is a nightmare*
  - *library standards*
- ***Java***: *significantly better*
  - *MUCH more rigid behaviour*
  - ***strictfp*** *for floats*
  - *some libraries still need care*
- *Other languages: case by case*

Compatibility Issues

- *Completely non-existing for Same-Executable Determinism* ✔
- *Often can be dealt with for Equivalence Testing and Replay-Based Regression Testing scenarios* ?
- *Extremely Nasty for Cross-Platform Determinism*
  – *can become hopeless for intensive floating-point calculations* ✘

Sources of Non-Determinism

Multithreading ✔

- *(Re)Actors*
- *Circular Logging*

System Calls ✔

- *Call Wrapping*
- *Pre-Calculation*
- *Non-Blocking Calls*

Risky Behaviours ✔

- *Under our Control*
- *Feasible to Avoid*

Compatibility Issues	***C/C++***	***Java***	***Others***
Same-Executable	✔	✔	✔
Equivalence Testing	?	✔	?
Cross-Platform	✗	?	?

Non-Issues

- *PRNG*
- *Text Logging/Tracing*
  – *time()/timeEnd() can call time() within without "call wrapping"*
- *Caching*
  – *either treated as a part of our deterministic program*
  – *or treated as residing "outside" of our deterministic program*
  – *may be useful to reduce size of serialised state*

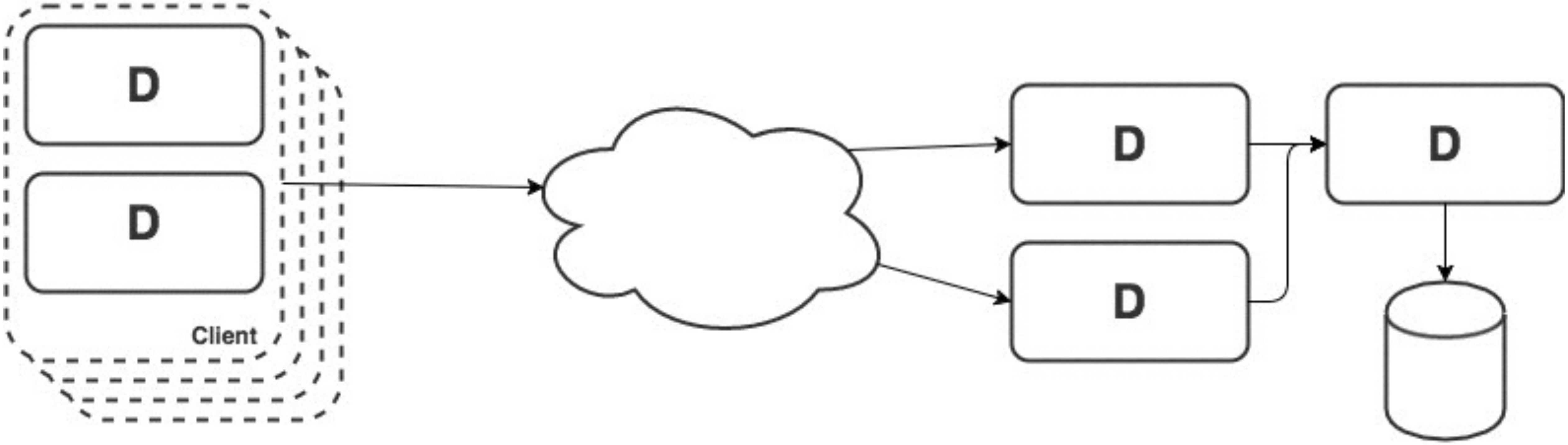
Part III. Building Interactive Distributed Systems

Interactive Distributed System

Properties:
- Distributed: built from components
– components are usually stateful
– communicate via messages
- Interactive
– typical response times are from single-digit milliseconds to single-digit seconds

Examples:
- Multiplayer Games
– including stock exchanges and auctions
- Any Reasonably Complex Device
– including laptops, smartphones, TVs, etc.
- Internet as a whole

Typical Structure



I've seen a system with thousands of (mostly) Deterministic Components on hundreds of Servers - and a few millions of (mostly) Deterministic Components running on hundreds of thousands of Client devices across the world.

The Problem

One of the biggest challenges for real-world Distributed Interactive Systems, is debugging and testing them.

For such systems, at least 80% of the bugs which have made it to production - are related to unusual sequences of incoming events.

Such bugs are especially nasty, as we cannot predict them in advance - and therefore cannot test them either.

The Solution

To address this problem, Deterministic Components help us with:

- improved overall testability
  – if we have a problem - we can reproduce it, and reproducible bug is a dead bug
  – bugs found in simulation testing

- Replay-Based Regression Testing

- production post-factum debugging
  – Over 80% of bugs fixed from first crash

Observed Result:
3x to 5x less downtime than industry average.

Making System Deterministic as a Whole

- *System built from Deterministic Components in not necessarily deterministic as a whole*
  - *unless special measures are taken - more often not than yes*
    - *most of the time - it is NOT a problem in practice*
- *Making the whole System deterministic is equivalent to establishing one single time for all the Components.*
  - *To do it - several methods exist, including CMS/LBTS, and "rewind" techniques similar to both financial "value date" and gaming "Server Rewind"*

I WANT YOU to go deterministic

Summary:

- ***Deterministic Components improve system quality significantly***, *via:*
  - *improved debugging*
  - *improved testing (including Replay-Based Regression Testing)*
  - *production post-factum debugging*

- ***Deterministic Components are achievable,*** *via:*
  - *(Re)Actors (or a reasonable facsimile)*
  - *Circular Logging*
  - *"Call Wrapping" and a few other techniques*

- ***WHAT ARE YOU WAITING FOR? ;-)***

Further Info

nobugs@ithare.com

OR

Chapter 5