An Overview of Distributed Debugging

Anant Narayanan

Advanced Topics in Distributed Systems
November 17, 2009

The Problem

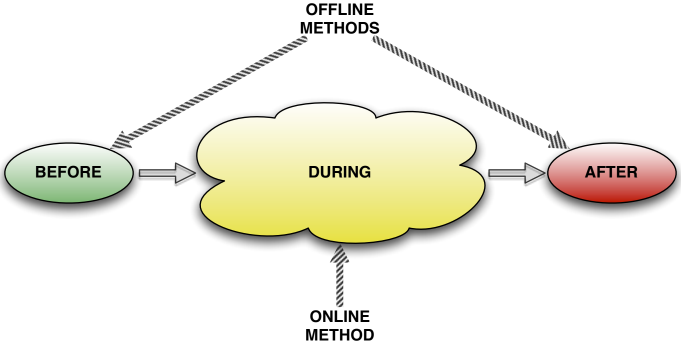
Anything that can go wrong will go wrong

Debugging is frustrating. Distributed debugging even more so!

Why is this hard?

- Errors are rarely reproducible
  - Non-determinism plays a big role in distributed systems
- Remote machines appear to crash more often!
- Interactions between several different components (possibly written in different languages) running on different computers are extremely intricate
- Communication is unreliable and asynchronous
- Existing debuggers are simply inadequate

Possible Approaches



Outline

1. After
   - Logging (liblog)
   - Pervasive debuggers
   - Time travel (TTVM)
2. Before
   - Model checking (MaceMC)
3. During
   - D^3S
   - CrystalBall

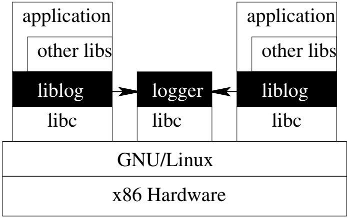
Logging

Example

```
printf("The value of x at node %d:  %d", nr, x);
```

- The most primitive form of debugging, we all do it!
- However, extremely difficult to capture all state, and thus can be used only for small bugs
- Won't it be a good idea to *automatically* capture and store all state information so we can analyze and possibly replay it at a later time?

Yes, it would!



- Intercepts all calls to `libc` using `LD_PRELOAD`
- Provides **continuous** logging with **deterministic** and **consistent** group replay in a **mixed** environment
- Integrates with `gdb` to provide central replay in a familiar environment

Challenges

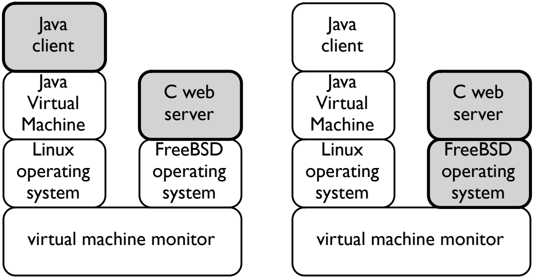
- Signals and Threads
  - User-level cooperative scheduler on top of OS scheduler
- Unsafe Memory Access
  - All `malloc` calls are effectively `calloc`
- Consistent Replay for UDP/TCP
  - Packets are annotated
- Finding Peers in a Mixed Environment
  - Local ports are tracked
  - Initialization with other `liblog` hosts occurs

Is liblog for you?

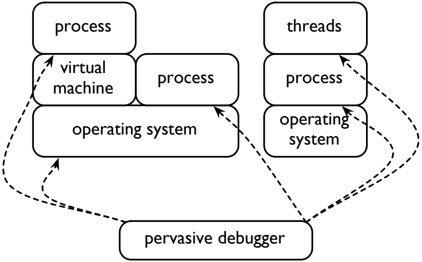
High disk usage; heterogenous systems and tight spin-locks disallowed; 16 byte per-message network overhead; and finally, limited consistency

A Pervasive Debugger

- Debuggers are unable to access all the state that we sometimes need because *it is just another program!*
- Debugging is usually either vertical or horizontal:



A Pervasive Debugger

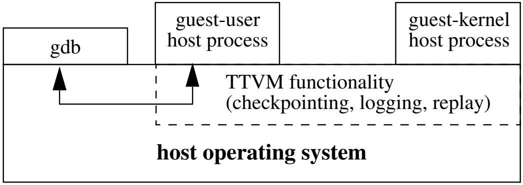


- Why are debuggers peers of the application being debugged rather than being placed in the *underlying* system?
- This architecture allows us to perform *both* vertical and horizontal debugging

Let's Look at an Application

- A Virtual Machine Monitor (VMM) is capable of monitoring and logging a lot more state than is possible by a userspace library!
- By running an application inside a VM, we are able to log not just CPU instructions, memory, network and disk I/O, but also **interrupts**, **clock** values, **signals**
- We can also log **byte-for-byte** network, memory and disk
  - Remember, device drivers can have bugs too!
- Time-traveling virtual machines take advantage of all this by using User Mode Linux (`UML`) and integrating with `gdb` to provide a unified, easy to use debugging environment

How This Works



- In addition to all the earlier mentioned state parameters, the system takes system checkpoints at regular intervals
- The host operating system, UML and gdb are modified to allow *time-travel* back to earlier checkpoints, replaying execution with breakpoints

Performance

Checkpointing every 25s adds just 4% overhead!

Model Checking

We've seen what tools we can use *after* a bug has been found, is there anything we can do *before* deploying an application?

- Model checkers, which basically perform state space exploration, can be used to gain confidence in a system
- `MaceMC` is one such model checker, tailored for verifying large distributed applications

Definition

Safety Property

A property that should **always** be satisfied

Liveness Property

A property that should **always** be **eventually** satisfied

Life, Death and the Critical Transition

- Each node is a state machine
- At each step in the execution, an event handler for a particular pending event at a node is called
- Thus, the entire system is to be represented as a giant state machine with specific event handlers defined
- Of course, liveness and safety properties are required by `MaceMC` to start the checks

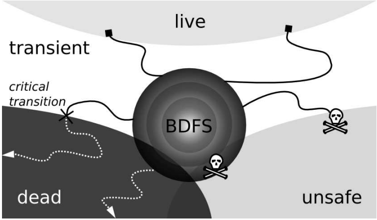
Definition

Critical transition

A transition from a **live** state to a **dead** state, from which a liveness property can *never* be satisfied

3 step process

1. Bounded depth-first search
2. Random walks
3. Isolating critical transitions



Is MaceMC for you?

Requires a concrete and theoretical model of your system. Existing code must be understood and represented as a state machine and properties! Too much work?

Debugging Deployed Solutions

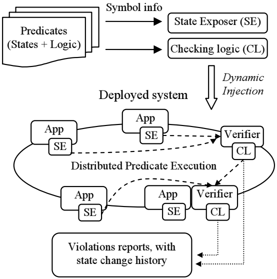
Because real debuggers run on a live, deployed system!

- Instead of verifying liveness properties in advance, why not let the system itself do a state space search for you?
- D^3S does exactly that by letting the developer specify *predicates* that are automatically verified by the system on-the-fly.

Key Challenge

Allowing developers to express predicates easily, verify those predicates in a distributed manner with minimal overhead, and **without disrupting the system**!

D^3S Architecture

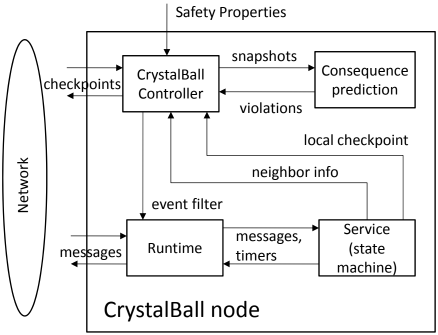


- Simple C++ API for specifying predicates and state
- Verifier and State exposer processes can be on different machines, allowing for **partitioned execution**
- Safety property violations are immediately logged, liveness properties after a *timeout*

Steering Deployed Solutions

- So, D^3S can *detect* property violations but can we do anything about it?
- CrystalBall attempts to give us an ultimate solution by gazing at the future and *steering* the application away from disaster!
- Many distributed application block on network I/O, let's use those free CPU cycles for some useful work...
  - Packet transmission is faster in simulation than in reality
- Can we stay *one-state-step* ahead at all times?

CrystalBall Architecture



- *Deep online debugging*: Property violations recorded
- *Execution Steering*: Avoids erroneous conditions reported

Challenges

- Specifying state and properties: Uses MaceMC
- Consistent snapshots: Only neighbors are involved
- Consequence prediction: Refined state-space search
- Steering without disruption: Filters rely on the distributed system handling "dropped" messages

How did it do?

Bugs found in RandTree, Chord, and Bullet' while in deep online debugging mode

As for execution steering, Bullet' ran for 1.4 hours with 121 inconsistent states that were never reached, no false negatives. When run on *Paxos*, inconsistencies at runtime were avoided between 74 and 89% of the time

Your Takeaways

Tools

liblog and **TTVM** at your disposal for debugging using the familiar gdb environment after a crash occurs
MaceMC model checking gives you theoretical confidence in your system before you deploy it

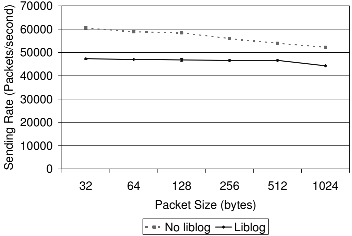
Systems

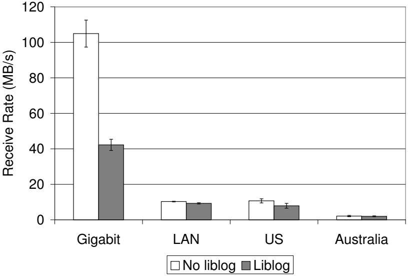
D^3S detects and logs the reason for property violations based on your specifications
CrystalBall can take this one step further and prevent your distributed system from executing towards bad states

Recommendation

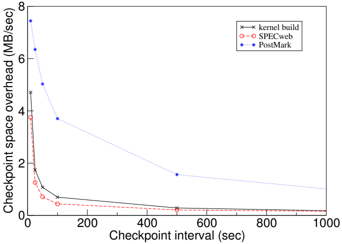
Use a combination of these tools and systems to make all your debugging problems go away!

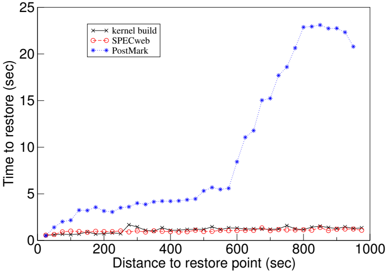
Performance: liblog



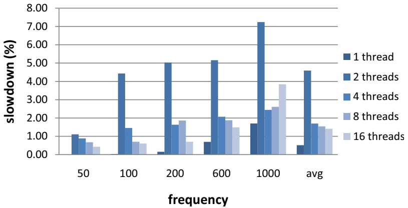


Performance: TTVM

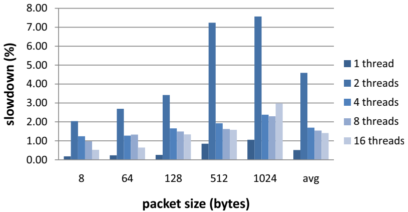




Performance: D^{3}S



(a) Slowdown with average packet size 390 bytes and different exposing frequencies.



(b) Slowdown with average frequency 347 /s and different exposing packet sizes.

Performance: CrystalBall