Replay Debugging for Distributed Systems

Dennis Geels, Gautam Altekar, Ion Stoica, Scott Shenker

What we've done

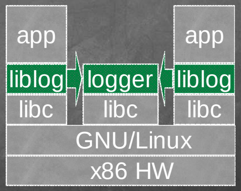
- liblog: lightweight logging and deterministic replay for distributed applications
- First tool that meets requirements. Also:
- No modifications to source or binary
- Support POSIX C/C++ apps
- No special hardware or kernel changes
- Familiar GDB interface

Why Another Debugger?

- Great distributed software being developed
- routing overlays, query processors BFT replication, DHTs
- More algorithms than users
- Distribution brings new bugs
- Current tools do not help deployed apps

Design: Logging

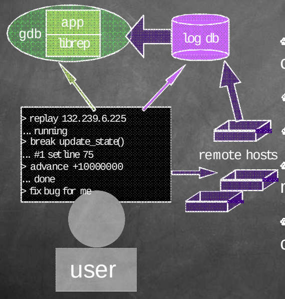
- Loads shared lib at runtime
- Intercepts libc calls
- Sends return values to logger daemon
- Logs, checkpoints compressed on disk
- Embed Lamport clocks in all network messages
- Incoming messages saved



What do we need?

- Requirements for debugging deployed applications:
- Independent logging: no central control
- log app execution for replay offsite
- Continuous operation: lightweight enough to leave debugging enabled.
- Consistent Group Replay: analyze distributed state together, without synchronized clocks
- Mixed Environment: not all peers will participate
- 3rd party clients, supporting services (DNS, db)

Design: Replay



```
> replay 132.239.6.225
... running
> break update_state()
... #1 set line 75
> advance +10000000
... done
> fix bug for me
```

- Central console coordinates replay
- Collect logs, ckpts
- UI: GDB++
- Replay arbitrary machines, times
- Virtual clocks allow consistent replay

Challenge: Threads

- Reading shared memory is nondeterministic
- Must reproduce contents or order of writes
- Same problem with `mmap`, signal handlers
- Solution: log and replay thread schedule
- Real challenge: no kernel support
- User-level locks serialize execution
- Blocking calls (e.g. `read`) run in background

Additional Challenges

- GDB support for migrated processes
- GDB support for multiple, synchronized processes
- Deterministic replay for programs with unsafe memory accesses
- Fast and durable logging

Challenge: User-level Annotations for TCP

- Must embed Lamport clocks at each send boundary
- Receiver need not respect send frames
- May not read more than requested by app (else block)
- ⊓ must recognize annotations on first byte
- Solution:
- Annotations precede each chunk of sent data
- 1-byte “magic”, clock, data chunk length
- 3-state machine: testing, reading tag, reading data
- loop between states until enough bytes read

Overhead

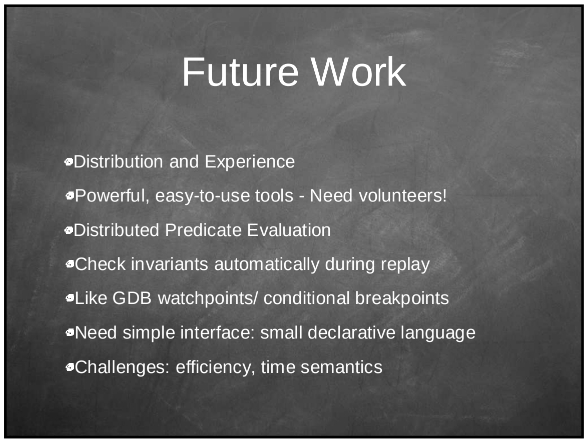
- Per-call wrapper latency: 1.5-2X (`sendto`)
- Fixed size UDP bandwidth: 2X
- 100 MB “empty” file transfer: 1.2MB logs
- 118MB logs for uncompressible data
- i3/chord daemon: 2.5 MB/hour
- Checkpoints: 10-20ms, 1 MB compressed

Challenge: Mixed Environment

- Message annotations confuse non-loggers
- Third-party clients
- Supporting protocols (DNS, ping, mysql)
- Federated/Partial deployment
- Solution: Integrated discovery service
- Query remote `logger` at well-known port
- Short timeouts, caching reduces impact

Experience

- Bugs found in I3/Chord and proxy:
- 2 broken assumptions about network
- 3 coding errors
- 2 proofs of weak bootstrap algorithms
- Used replay to debug debugger:
- Message tags, missing `libc` wrappers, uninitialized memory reads by programs
- Started manually injecting bugs into I3



Future Work

- Distribution and Experience
- Powerful, easy-to-use tools - Need volunteers!
- Distributed Predicate Evaluation
- Check invariants automatically during replay
- Like GDB watchpoints/ conditional breakpoints
- Need simple interface: small declarative language
- Challenges: efficiency, time semantics

Thank you