Building diagnosable distributed systems

Petros Maniatis
Intel Research Berkeley

ICSI – Security Crystal Ball
...and how that’s going to solve all of our (security) problems

Petros Maniatis
Intel Research Berkeley

ICSI – Security Crystal Ball
Why distributed systems are hard

In every application, organizing distributed resources has unique needs
• Low latency, high bandwidth, high reliability, tolerance to churn, anonymity, long-term preservation, undeniable transactions, ...

So, for every application one must
• Figure out right properties
• Get the algorithms/protocols
• Implement them
• Tune them
• Test them
• Debug them
• Repeat
Why distributed systems are hard (among other things)

In every application, organizing distributed resources has unique needs

- Low latency, high bandwidth, high reliability, tolerance to churn, anonymity, long-term preservation, undeniable transactions, ...

So, for every application one must

- Figure out right properties
- Get the algorithms/protocols
- Implement them
- Tune them
- Test them
- Debug them
- Repeat

- No global view
- Bad algorithm choice
- Incorrect implementation
- Psychotic timeouts
- Partial failures
- Biased introspection
- Homicidal boredom
Diagnostic vs. Diagnosable Systems

**Diagnostic** is good
- Complex query processing
- Probes into distributed system exporting data to be queried
- Nice properties
  - Privacy, efficiency, timeliness, consistency, accuracy, verifiability

**Diagnosable** is even better
- Information flow can be observed
- Control flow can be observed
- Observations can be related to system specification
  - no need for “translation”

If you are really lucky, you can get both
Strawman Design: P2
(Intel, UCB, Rice, MPI)

Intended for user-level distributed system design, implementation, monitoring

Specify high-level system properties
• Abstract topology
• Transport properties

Currently: in a variant of Datalog
• Looks like event-condition-action rules
• Location specifiers place conditions in space
• Rendezvous happens under the covers
Strawman Design: P2

Intended for user-level distributed system design, implementation, monitoring

Specify high-level system properties
• Abstract topology
• Transport properties

Currently: in a variant of Datalog
• Looks like event-condition-action rules
• Location specifiers place conditions in space
• Rendezvous happens under the covers

response@Req(Key, SAddr1) :-
lookup@NAddr(Req, Key),
node@NAddr(NodeID),
succ@Naddr(NextID, SAddr),
succ@SAddr(NextNextID, SAddr1),
Key in (NodeID, Succ].

In English:
• Return a response to Req
• When node NAddr receives lookup for key Key
• And that key lies between NAddr’s identifier and that of its successor’s successor
Strawman Design: P2 Specification to Implementation

Automatically compile specification into a dataflow graph

• Elements are little-functionality operators
  – Filters, multiplexers, loads, stores

• Elements are interlinked with asynchronous data flows
  – Data unit: a typed, flat tuple

• Reversible optimizations happen below
Strawman Design: P2 Diagnosis

Flows are observable
• Treat them as any other data stream, store, query them...

Element state transitions are observable
• Similarly, store them, query them, export them...
Execution is observable
- Whenever a rule creates an output, I can log the inputs that caused that output
- Generate a causal stream at the granularity of the user’s specification steps
What we know we can do (e.g., in overlays)

Monitoring invariants: a distributed watchpoint
- “A node’s in-degree is greater than k”
- “Routing is inconsistent”
- “A node oscillates into and out of a neighborhood’s routing tables”

Execution tracing at “pseudo-code” granularity: logical stepping
- Obtain the causality graph of a failed lookup (what steps led to failure)
- Find failed lookups with state oscillations in its history (filter failure cause)
- Install additional sensors where state oscillations usually cause lookup failure
- Mining of causality graph on-line

Complex programmatic system views
- Run a consistent snapshot on the state of the system
- Query the consistent snapshot on stable properties
- Take system checkpoints
Opportunities
What we’d love to be able to do but don’t quite know exactly how yet

Small set of basic components
• Relational operators, flow operators, network operators
• Should be able to formally validate them
• Then compose to statically validate entire graph (termination, information flow, etc.)

Verifiable dataflow graph execution
• Attest all dataflow elements in isolation
• Attest resulting dataflow graph

Undeniable causality graphs
• Tamper-evident logs per dataflow element
• (Automatically) composable proofs of compliant execution of dataflow graph

Smaller combinatorial problems:
• Chord overlay: ~10000 individual lines of code in C++
• Compare that to our P2Chord: ~300 elements, ~12 reusable types, + engine
What we’d love to be able to do but don’t quite know exactly how yet

What might be a useful checked invariant?
• Known protocol weakness we don’t know how to fix
• Design a distributed query checking for it
• When it triggers, abort, work less slowly, notify authorities, give random answer, ...

Graph rewrites that preserve specific properties
• Information declassification (look at Saberfeld et al)
• Routing path decorrelation

Who cares about verifiable, user-level distributed applications?
• Monitoring within disparate organizations

Who cares about undeniable, user-level distributed applications?
• Same, when liability is involved (e.g., anti-framing)
What do we have here?

Constrained programming-execution environment
• Typed information flow
• Very modular design
• Small set of functional modules can build large set of combinations
• Only reversible optimizations allowed

But we’re nowhere near solving the malicious code understanding prob
• Can we afford to take the performance hit?
• Fairly coarse granularity
• Can you write a general-purpose Distributed FS on this? Hmm...

Thesis: Defining a usable (if not complete) programming paradigm that allows “effortless” due diligence might improve the situation
Thank You!

http://berkeley.intel-research.net/maniatis/