Declarative Overlays

Petros Maniatis
joint work with Tyson Condie, David Gay, Joseph M. Hellerstein, Boon Thau Loo, Raghu Ramakrishnan, Sean Rhea, Timothy Roscoe, Atul Singh, Ion Stoica
IRB, UC Berkeley, U Wisconsin, Rice
Overlays Everywhere...

“Overlay”: the routing and message forwarding component of any self-organizing distributed system
Overlays Everywhere...

- Many examples:
  - Internet Routing, multicast
  - Content delivery, file sharing, DHTs, Google
  - Microsoft Exchange (for load balancing)
  - Tibco (technology bridging)
- Overlays are a fundamental tool for repurposing communication infrastructures
  - Get a bunch of friends together and build your own ISP (Internet evolvability)
  - You don’t like Internet Routing? Make up your own rules (RON)
  - Paranoid? Run a VPN in the wide area
  - Intrusion detection with friends (FTN, Polygraph)
  - Have your assets discover each other (iAMT)
Overlays Everywhere...

- Many examples:
  - Internet Routing, multicast
  - Content delivery, file sharing, DHTs, Google
  - Microsoft Exchange (for load balancing)

Distributed systems innovation needs overlays

- You don’t like Internet Routing? Make up your own rules (RON)
- Paranoid? Run a VPN in the wide area
- Intrusion detection with friends (FTN, Polygraph)
- Have your assets discover each other (iAMT)
If only it weren’t so hard

- In theory
  - Figure out right properties
  - Get the algorithms and protocols
  - Implement them
  - Tune them
  - Test them
  - Debug them
  - Repeat
If only it weren’t so hard

- In theory
  - Figure out right properties
  - Get the algorithms and protocols
  - Implement them
  - Tune them
  - Test them
  - Debug them
  - Repeat

- But in practice
If only it weren’t so hard

• In theory
  • Figure out right properties
  • Get the algorithms and protocols
  • Implement them
  • Tune them
  • Test them
  • Debug them
  • Repeat

• But in practice
  • No global view
  • Wrong choice of algorithms
  • Incorrect implementation
  • Psychotic timeouts
  • Partial failures
  • Impaired introspection
  • Homicidal boredom
If only it weren’t so hard

- In theory
  - Figure out right properties
  - Get the algorithms and protocols

- But in practice
  - No global view
  - Wrong choice of algorithms

It’s hard enough as it is
Do I also need to reinvent the wheel every time?

- Debug them
- Repeat

- Impaired introspection
- Homicidal boredom
Our Goal

- Make overlay development more accessible to developers of distributed applications
  - Specify overlay at a high-level
  - Automatically translate specification into executable
  - Hide everything they don’t want to touch
  - Enjoy performance that is *good enough*
- Do for networked systems what the relational revolution did for databases
Enter P2: Semantics

- Distributed state
  - Distributed soft state in relational tables, holding tuples of values
    - route \((S, D, H)\)
  - Non-stored information passes around as *event tuple streams*
    - message \((X, D)\)
Enter P2: Semantics

- Distributed state
  - Distributed soft state in relational tables, holding tuples of values
    - route \((S, D, H)\)
  - Non-stored information passes around as event tuple streams
    - message \((X, D)\)

- Overlay specification in declarative logic language (OverLog)
  - \(<\text{head}> : - <\text{precondition1}>, <\text{precondition2}>, \ldots, <\text{preconditionN}>.\)
  - Location specifiers \(@\text{Loc place} \) individual tuples at specific nodes
  - message@H(H, D) : - route@S(S, D, H), message@S(S, D).\)
Enter P2: Semantics

- Distributed state
  - Distributed soft state in relational tables, holding tuples of values
    - route \((S, D, H)\)
  - Non-stored information passes around as event tuple streams
    - message \((X, D)\)

- Overlay specification in declarative logic language (OverLog)
  - \(<\text{head}> :- <\text{precondition1}>, <\text{precondition2}>, \ldots, <\text{preconditionN}>.\)
  - Location specifiers \(\text{@Loc place}\) individual tuples at specific nodes
  - message\(\text{@H(H, D)} :- \text{route@S(S, D, H)}, \text{message@S(S, D)}.\)
Enter P2: Semantics

- Overlay specification in declarative logic language (OverLog)
  - \(<\text{head}> \leftarrow <\text{precondition1}>, <\text{precondition2}>, \ldots, <\text{preconditionN}>\).
  - Location specifiers \(@\text{Loc} \text{ place}\) individual tuples at specific nodes
  - message@\text{H}(H, D) \leftarrow \text{route}@\text{S}(S, D, H), \text{message}@\text{S}(S, D).

message@a(a, z)

(a, y, c)
(a, z, r)
(a, z, t)
Enter P2: Semantics

- Overlay specification in declarative logic language (OverLog)
  
  - `<head> :- <precondition1>, <precondition2>, ... , <preconditionN>`.
  
  - Location specifiers `@Loc place` individual tuples at specific nodes
  
  - `message@H(H, D) :- route@S(S, D, H), message@S(S, D)`.
Enter P2: Semantics

- Overlay specification in declarative logic language (OverLog)
  - `<head> :- <precondition1>, <precondition2>, ..., <preconditionN>.
  - Location specifiers `@Loc place` individual tuples at specific nodes
  - `message@H(H, D) :- route@S(S, D, H), message@S(S, D).

```
message@r(r, z)
message@t(t, z)
```

Research
Enter P2: Semantics

- Overlay specification in declarative logic language (OverLog)
  - `<head> :- <precondition1>, <precondition2>, ... , <preconditionN>`.
  - Location specifiers `@Loc place` individual tuples at specific nodes
  - `message@H(H, D) :- route@S(S, D, H), message@S(S, D)`.

Research

Petros Maniatis, IRB
Enter P2: Dataflow

- Specification automatically translated to a dataflow graph
- C++ dataflow elements (akin to Click elements)
- Implement
  - relational operators (joins, selections, projections)
  - flow operators (multiplexers, demultiplexers, queues)
  - network operators (congestion control, retry, rate limitation)
- Interlinked via asynchronous push or pull typed flows
  - Pull carries a callback from the puller in case it fails
  - Push always succeeds, but halts subsequent pushes
- Execution engine runs the dataflow graph
  - Simple FIFO event scheduler (a la libasync) for I/O, alarms, deferred execution, etc.
Enter P2: Dataflow

- Specification automatically translated to a dataflow graph
  - C++ dataflow elements (akin to Click elements)
  - Implement
    - relational operators (joins, selections, projections)

A distributed query processor to maintain overlays

- Push always succeeds, but halts subsequent pushes
- Execution engine runs the dataflow graph
  - Simple FIFO event scheduler (a la libasync) for I/O, alarms, deferred execution, etc.
Example: Ring Routing

- Every node has an address (e.g., IP address) and an identifier (large random)
- Every object has an identifier
- Order nodes and objects into a ring by their identifiers
- Objects “served” by their successor node
- Every node knows its successor on the ring
- To find object $K$, walk around the ring until I locate $K$'s immediate successor node
Example: Ring Routing

- How do I find the responsible node for a given key k?

- \text{n.lookup(k)}
  
  \text{if k in (n, n.successor)}
  
  \text{return n.successor}

  \text{else}
  
  \text{return n.successor. lookup(k)}
Ring State

- \( n.\text{lookup}(k) \)
  
  if \( k \) in \( (n, n.\text{successor}) \)
  
  return \( n.\text{successor} \)
  
  else
  
  return \( n.\text{successor}.\text{lookup}(k) \)

- Node state tuples
  
  - node(NAddr, N)
  
  - successor(NAddr, Succ, SAddr)

- Transient event tuples
  
  - lookup (NAddr, Req, K)
Pseudocode to OverLog

- \( n\.lookup(k) \)
  
  ```
  if k in (n, n.successor] 
    return n.successor 
  else 
    return n.successor. lookup(k) 
  ```

- Node state tuples
  - node\((N\text{Addr}, N)\)
  - successor\((N\text{Addr}, \text{Succ}, S\text{Addr})\)

- Transient event tuples
  - lookup \((N\text{Addr}, \text{Req}, K)\)

R1 response \((\text{Req}, K, S\text{Addr})\):

- lookup \((N\text{Addr}, \text{Req}, K)\),
- node \((N\text{Addr}, N)\),
- succ \((N\text{Addr}, \text{Succ}, S\text{Addr})\),
- \( K \) in \((N, \text{Succ})\).
Pseudocode to OverLog

- `n.lookup(k)`
  
  if k in (n, n.successor)
  
  return n.successor

- else
  
  return n.successor. lookup(k)

- Node state tuples
  
  - node(NAddr, N)
  
  - successor(NAddr, Succ, SAddr)

- Transient event tuples
  
  - lookup (NAddr, Req, K)

R1 response (Req, K, SAddr) :-
  
  lookup (NAddr, Req, K),

  node (NAddr, N),

  succ (NAddr, Succ, SAddr),

  K in (N, Succ).

R2 lookup (SAddr, Req, K) :-
  
  lookup (NAddr, Req, K),

  node (NAddr, N),

  succ (NAddr, Succ, SAddr),

  K not in (N, Succ).
Location Specifiers

- `n.lookup(k)`
  
  if \( k \) in \( (n, n\text{.successor}) \)
  
  return \( n\text{.successor} \)
  
  else
  
  return \( n\text{.successor}.\) lookup\( (k) \)

- Node state tuples
  
  - `node(NAddr, N)`
  
  - `successor(NAddr, Succ, SAddr)`

- Transient event tuples
  
  - `lookup (NAddr, Req, K)`

R1 response @Req(Req, K, SAddr) :-

  lookup@NAddr(NAddr, Req, K),

  node@NAddr(NAddr, N),

  succ@NAddr(NAddr, Succ, SAddr),

  K in (N, Succ).

R2 lookup@SAddr(SAddr, Req, K) :-

  lookup@NAddr(NAddr, Req, K),

  node@NAddr(NAddr, N),

  succ@NAddr(NAddr, Succ, SAddr),

  K not in (N, Succ).
From OverLog to Dataflow

R1 response@R(R, K, SI) :- lookup@NI(NI, R, K),
        node@NI(NI, N), succ@NI(NI, S, SI), K in (N, S).

R2 lookup@SI(SI, R, K) :- lookup@NI(NI, R, K),
        node@NI(NI, N), succ@NI(NI, S, SI), K not in (N, S).
From OverLog to Dataflow

R1 response@R(R, K, S_l) : - lookup@NI(NI, R, K), node@NI(NI, N), succ@NI(NI, S, S_l), K in (N, S).

R2 lookup@SI(S_l, R, K) : - lookup@NI(NI, R, K), node@NI(NI, N), succ@NI(NI, S, S_l), K not in (N, S).
From OverLog to Dataflow

R1 response@R(R, K, SI) : - lookup@NI(NI, R, K), node@NI(NI, N), succ@NI(NI, S, SI), K in (N, S).

R2 lookup@SI(SI, R, K) : - lookup@NI(NI, R, K), node@NI(NI, N), succ@NI(NI, S, SI), K not in (N, S).
R1 response@R(R, K, SI) : - lookup@NI(NI, R, K), node@NI(NI, N), succ@NI(NI, S, SI), K in (N, S).

R2 lookup@SI(SI, R, K) :- lookup@NI(NI, R, K), node@NI(NI, N), succ@NI(NI, S, SI), K not in (N, S).
From OverLog to Dataflow

R1 \text{response@R}(R, K, SI) : - \text{lookup@NI}(NI, R, K), \node@NI(NI, N), \text{succ@NI}(NI, S, SI), K \in (N, S).

R2 \text{lookup@SI}(SI, R, K) : - \text{lookup@NI}(NI, R, K), \node@NI(NI, N), \text{succ@NI}(NI, S, SI), K \not\in (N, S).
R1 \text{ response}@R(R, K, SI) : - \text{ lookup}@NI(NI, R, K), \node@NI(NI, N), \text{ succ}@NI(NI, S, SI), K \text{ in } (N, S].}

R2 \text{ lookup}@SI(SI, R, K) : - \text{ lookup}@NI(NI, R, K), \node@NI(NI, N), \text{ succ}@NI(NI, S, SI), K \text{ not in } (N, S].
From OverLog to Dataflow

R1 \text{response@R}(R, K, SI) : - \text{lookup@NI}(NI, R, K), node@NI(NI, N), succ@NI(NI, S, SI), K \text{ in } (N, S].

R2 \text{lookup@SI}(SI, R, K) : - \text{lookup@NI}(NI, R, K), node@NI(NI, N), succ@NI(NI, S, SI), K \text{ not in } (N, S].
From OverLog to Dataflow

R1 response@R(R, K, SI) :- lookup@NI(NI, R, K), node@NI(NI, N), succ@NI(NI, S, SI), K in (N, S).

R2 lookup@SI(SI, R, K) :- lookup@NI(NI, R, K), node@NI(NI, N), succ@NI(NI, S, SI), K not in (N, S).
From OverLog to Dataflow

- One *rule strand* per OverLog rule
- Rule order is immaterial
- Rule strands could execute in parallel
A Bit of Chord
Chord on P2

- Full specification of ToN Chord
  - Multiple successors
  - Stabilization
  - Failure recovery
  - Optimized finger maintenance
- 46 OverLog rules
  - (1 USLetter page, 10pt font)
- How do we know it works?
  - Same high-level properties
    - Logarithmic overlay diameter
    - Logarithmic state size
    - Consistent routing with churn
- "Comparable" performance to hand-coded implementations
Lookup length in hops (no churn)
Maintenance bandwidth (no churn)
Lookup Latency (no churn)
Lookup Latency (with churn)
Lookup Consistency (with churn)

- Consistent fraction: size fraction of largest result cluster
  - k lookups, different sources, same destination
Maintenance bandwidth (churn)
But Still a Research Prototype

- Bugs still creep up (algorithmic logic / P2 impl.)
  - Multi-resolution system introspection
- Application-specific network tuning, auto or otherwise still needed
  - Component-based reconfigurable transports
- Logical duplications ripe for removal
  - Factorizations and Cost-based optimizations
1. System Introspection

- Two unique opportunities
  - Transparent execution tracing
  - A distributed query processor on all system state
Execution Tracing and Logging

- Execution tracing/logging happens externally to system specification
  - At “pseudo-code” granularity: logical stepping
  - Why did rule R7 trigger? Under what preconditions?
  - Every rule execution (input and outputs) is exported as a table
    - ruleExec(Rule, InTuple, OutTuple, OutNode, Time)
  - At dataflow granularity: intermediate representation stepping
    - Why did that tuple expire? What dropped from that queue?
    - Every dataflow element execution exported as a table, flows tapped and exported
      - queueExec(...), roundRobinExec(...), ...

- Transparent logging by the execution engine
  - No need to insert printf’s and hope for the best
  - Can traverse execution graph for particular system events
    - Its preconditions, and their preconditions, and so on across the net
Distributed Query Processing

- Once you have a distributed query processor, lots of things fall off the back of the truck
  - Overlay invariant monitoring: a distributed watchpoint
    - “What’s the average path length?”
    - “Is routing consistent?”
  - Pattern matching on distributed execution graph
    - “Is a routing entry gossiped in a cycle?”
    - “How many lookup failures were caused by stale routing state?”
    - “What are the nodes with best-successor in-degree > 1?”
    - “Which bits of state only occur when a lookup fails somewhere?”
  - Monitoring disparate overlays / systems together
    - “When overlay A does this, what is overlay B doing?”
    - “When overlay A does this, what is the network, average CPU, … doing?”
2. Reconfigurable Transport

- New lease on life of an old idea!
- Dataflow paradigm thins out layer boundaries
  - Mix and match transport facilities (retries, congestion control, rate limitation, buffering)
  - Spread bits of transport through the application to suit application requirements
    - Move buffering before computation
    - Move retries before route selection
    - Use single congestion control across all destinations
- Express transport spec at high-level
  - “Packetize all msgs to same dest together, but send acks separately”
  - “Packetize updates but not acks”
3. Automatic Optimization

- Optimize within rules
  - Selects before joins, join ordering
- Optimize across rules & queries
  - Common “subexpression” elimination
- Optimize across nodes
  - Send the smallest relation over the network
  - Caching of intermediate results
- Optimize scheduling
  - Prolific rules before deadbeats
What We Don’t Know (Yet)

- The limits of first-order logic
  - Already pushing through to second-order, to do introspection
  - Can be awkward to translate inherently imperative constructs, etc. if-then-else / loops
- The limits of the dataflow model
  - Control vs. data flow
  - Can we eliminate (most) queues? If not, what’s the point?
  - Can we do concurrency control for parallel execution?
- The limits of “automation”
  - Can we (ever) do better than hand-coded implementations? Does it matter?
  - How good is good enough?
  - Will designers settle for auto-generation? DBers did, but this is a different community
- The limits of static checking
  - Can we keep the semantics simple enough for existing checks (termination, safety, …) to still work automatically?
Related Work

• Early work on executable protocol specification
  • Esterel, Estelle, LOTOS (finite state machine specs)
  • Morpheus, Prolac (domain-specific, OO)
  • RTAG (grammar model)

• Click
  • Dataflow approach for routing stacks
  • Larger elements, more straightforward scheduling

• Deductive / active databases
Summary

- Overlays enable distributed system innovation
- We’d better make them easier to build, reuse, understand
- P2 enables
  - High-level overlay specification in OverLog
  - Automatic translation of specification into dataflow graph
  - Execution of dataflow graph
- Explore and Embrace the trade-off between fine-tuning and ease of development
- Get the full immersion treatment in our papers at SIGCOMM and SOSP ‘05
Questions
(a few to get you started)

- Who cares about overlays?
- Logic? You mean Prolog? Eeew!
- This language is really ugly. Discuss.
- But what about security?
- Is anyone ever going to use this?
- Is this as revolutionary and inspired as it looks?