P2: Implementing Declarative Overlays

Timothy Roscoe
Boon Thau Loo, Tyson Condie, David Gay, Joseph M. Hellerstein, Petros Maniatis, Ion Stoica
Intel Research at Berkeley
UC Berkeley
Overlays: a broad view

“Overlay”: the routing and message forwarding component of *any* non-trivial distributed system
Overlays Everywhere...

- Many examples:
  - Internet Routing, multicast
  - Content delivery, file sharing, DHTs, Google
  - Microsoft Exchange
  - Tibco (technology interoperation)

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 Freenet
- Intrusion detection with friends (DDI, Polygraph)
- Have your assets discover each other (iAMT)

Distributed systems innovation needs overlays.
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
  - Impaired introspection
- Repeat
  - Homicidal boredom
  - Next to no debug support
Our Goal

- Make network development more accessible to developers of distributed applications
  - Specify network 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 SQL and the relational model did for databases
The argument:

- The set of routing tables in a network represents a *distributed data structure*
- The data structure is characterized by a set of ideal *properties* which define the network
  - Thinking in terms of structure, not protocol
- *Routing* is the process of maintaining these properties in the face of changing ground facts
  - Failures, topology changes, load, policy…
Routing as Query Processing

- In database terms, the routing table is a view over changing network conditions and state.

- Maintaining it is the domain of distributed continuous query processing.

- Not merely an analogy: We have implemented a general routing protocol engine as a query processor.
Two directions

1. Declarative expression of Internet Routing protocols
   • Loo et. al., ACM SIGCOMM 2005

2. Declarative *implementation* of overlay networks
   • Loo et. al., ACM SOSP 2005
   • The focus of this talk (and my work)
P2: A Declarative Overlay Engine

- 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)
  - head ::= precondition1, precondition2, ..., preconditionN.
  - Location specifiers place tuples at specific nodes
    - message@X(H, D) :- route@S(S, D, H), message@S(S, D).

(a, x, c) (a, z, f) (a, z, t)
P2 Dataflow

- Overlog automatically translated to dataflow graph
  - C++ dataflow elements (similar to Click elements)
  - Implements:
    - relational operators (joins, selections, projections)
    - flow operators (multiplexers, demultiplexers, queues)
    - network operators (congestion control, retry, rate limits)
  - Interlinked via asynchronous push or pull typed flows
- Engine executes dataflow graph at runtime

A distributed query processor to maintain overlays
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?

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

  else
  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(\( N\text{Addr}, N \))
  - successor(\( N\text{Addr}, \text{Succ}, S\text{Addr} \))

- Transient event tuples
  - lookup (\( N\text{Addr}, \text{Req}, K \))
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(NAddr, N)`
  - `successor(NAddr, Succ, SAddr)`

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

  ```
  response@Req (Req, K, SAddr) :-
  lookup@NAddr (NAddr, Req, K),
  node (NAddr, N),
  succ (NAddr, Succ, SAddr),
  K in (N, Succ].
  ```
**Pseudocode to OverLog**

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

- **Node state tuples**
  
  - Node \((\text{NAddr}, N)\)
  
  - Successor \((\text{NAddr}, \text{Succ}, \text{SAddr})\)

- **Transient event tuples**
  
  - lookup \((\text{NAddr}, \text{Req}, K)\)
  
  response@\(\text{Req} \) (\(\text{Req}, K, \text{SAddr}\)) :-
    lookup@\(\text{NAddr} \) (\(\text{NAddr}, \text{Req}, K\)),
    node (\(\text{NAddr}, N\)),
    succ (\(\text{NAddr}, \text{Succ}, \text{SAddr}\)),
    \( K \) in \((N, \text{Succ})\).

  lookup@\(\text{SAddr} \) (\(\text{SAddr}, \text{Req}, K\)) :-
    lookup@\(\text{NAddr} \) (\(\text{NAddr}, \text{Req}, K\)),
    node (\(\text{NAddr}, N\)),
    succ (\(\text{NAddr}, \text{Succ}, \text{SAddr}\)),
    \( K \) not in \((N, \text{Succ})\).
**Location Specifiers**

- `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**

```
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**

```
lookup@SAddr(SAddr, Req, K) :-
  lookup@NAddr(NAddr, Req, K),
  node@NAddr(NAddr, N),
  succ@NAddr(NAddr, Succ, SAddr),
  K not in (N, Succ].
```
Implementation: From OverLog to Dataflow

- Traditional problem in databases
- Turn logic into relational algebra
  - Joins, projections, selections, aggregations, etc.
From OverLog to Dataflow

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

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

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

lookup

Join
lookup.NI == node.NI

Join
lookup.NI == succ.NI

Select
K in (N, S]

NI, R, K, N, S, SI
K 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 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 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

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, 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

![Diagram showing two rule strands, Rule R1 and Rule R2, with nodes and succs connected by lookup arrows.]
From OverLog to Dataflow

Rule R1

Rule R2

...
Implementation

- Elements are C++ objects
  - Reference-counted immutable tuples
- Fast tuple hand-off
  - ~50 ia32 instructions, ~300 cycles
- Currently single-threaded
  - Select loop, timers, etc.
- Element state stored in tables
  - C.f. database catalogues: reuse data model wherever appropriate
- Conventional Bison/Flex parser
It actually works.

- For instance, we implemented Chord in P2
  - Popular distributed hash table
  - Complex overlay
  - Dynamic maintenance
- 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
Key point: remarkably concise overlay specification

- Full specification of Chord overlay, including
  - Failure recovery
  - Multiple successors
  - Stabilization
  - Optimized maintenance
- 44 OverLog rules
- And it runs!
Comparison: MIT Chord in C++
Lookup length in hops

![Graph showing frequency vs hop count for different node counts: 100 nodes, 300 nodes, 500 nodes.](image)
Maintenance bandwidth
(comparable with MIT Chord)
Latency without churn
Latency under churn

Compare with Bamboo non-adaptive timeout figures...
Consistency under churn

![Graph showing consistency under churn]

- CDF
- Consistent fraction
- Consistency threshold
The story so far:

- Can specify overlays as continuous queries in a logic language
- Compile to a graph of dataflow elements
- Efficiently execute graph to perform routing and forwarding
- Overlays exhibit similar performance characteristics

But …

Once you have a distributed query processor, lots of things fall off the back of the truck…
What else does this buy you?

**Introspection** (w/ Atul Singh, Rice)

- Overlay invariant monitoring: a *distributed watchpoint*
  - “What’s the average path length?”
  - “Is routing consistent?”

- Execution tracing at “pseudo-code” granularity: *logical stepping*
  - Why did rule R7 trigger?

- … and at dataflow granularity: *intermediate representation stepping*
  - Why did that tuple expire?

- Great way to do distributed debugging and logging
  - In fact, we use it and have found a number of bugs…
What else does this buy you?

2. Transport reconfiguration

- 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
- Automatically!

![Diagram](image-url)
In fact, a rich seam for future research...

- Reconfigurable transport protocols
- Debugging and logging support
- The “right” language – global invariants
  - Use distributed joins as abstraction mechanism
- Optimization techniques
  - Inc. multiquery optimization
- Monitoring other distributed systems and networks
  - Evolve towards more general query processor?
  - PIER heritage returns
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 paper in SOSP ’05, code release imminent
Thanks! 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?