P2: Implementing Declarative Overlays

Timothy Roscoe
Boon Thau Loo, Tyson Condie, Petros Maniatis, Ion Stoica, David Gay, Joseph M. Hellerstein,
Intel Research Berkeley
and U. C. 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 is all about 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 ultimate 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.

• Dataflow elements provide an implementation model for queries

• Overlays can be written in a high-level query language
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)
Data model

- Relational data: tuples and relations
- Two kinds of relation:
  - 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)
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?
• \( n.\text{lookup}(k) \)
  
  if \( k \) in \( (n, n.\text{successor}) \)
  
  return \( n.\text{successor} \)
  
  else
  
  return \( n.\text{successor} . \text{lookup}(k) \)
Ring State

- 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(Addr, Req, K)
  - response(Addr, K, Owner)
Pseudocode as a query

- `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 ( Addr, Req, K )`
  
  - `response( Addr, K, Owner )`

send `response( Req, K, SAddr )` to Req

where `lookup( NAddr, Req, K )` @ NAddr

and `node ( NAddr, N )`,

and `succ ( NAddr, Succ, SAddr )`,

and K in ( N, Succ ),
Pseudocode as a query

- **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 (Addr, Req, K)
  - response(Addr, K, Owner)

send **response( Req, K, SAddr )** to Req
where **lookup( NAddr, Req, K ) @ NAddr**
and **node ( NAddr, N ),**
and **succ ( NAddr, Succ, SAddr ),**
and **K in [ N, Succ ],**

send **lookup( Req, K, SAddr )** to SAddr
where **lookup( NAddr, Req, K ) @ NAddr**
and **node ( NAddr, N ),**
and **succ ( NAddr, Succ, SAddr ),**
and **K not in [ N, Succ ],**
Implementation: From query model to dataflow

- Traditional problem in databases
- Turn the logic into relational algebra
  - Joins, projections, selections, aggregations, etc.
- Implement as graph of software dataflow elements
  - C.f. Click, PIER, etc.
  - Tuples flow through graph
- Execute this graph to maintain overlay
From query to dataflow

send response( Req, K, SAddr ) to Req
   where lookup( NAddr, Req, K ) @ NAddr & node ( NAddr, N )
         & succ ( NAddr, Succ, SAddr ) & K in ( N, Succ ]

send lookup( Req, K, SAddr ) to SAddr
   where lookup( NAddr, Req, K ) @ Naddr & node ( NAddr, N )
         & succ ( NAddr, Succ, SAddr ) & K not in ( N, Succ ]
From query to dataflow

send response( Req, K, SAddr ) to Req
where lookup( NAddr, Req, K ) @ NAddr & node( NAddr, N )
& succ( NAddr, Succ, SAddr ) & K in ( N, Succ )

send lookup( Req, K, SAddr ) to SAddr
where lookup( NAddr, Req, K ) @ Naddr & node( NAddr, N )
& succ( NAddr, Succ, SAddr ) & K not in ( N, Succ )
From query to dataflow

send response( Req, K, SAddr ) to Req
where lookup( NAddr, Req, K ) @ NAddr & node( NAddr, N )
& succ( NAddr, Succ, SAddr ) & K in ( N, Succ )

send lookup( Req, K, SAddr ) to SAddr
where lookup( NAddr, Req, K ) @ NAddr & node( NAddr, N )
& succ( NAddr, Succ, SAddr ) & K not in ( N, Succ )
From query to dataflow

send response(Req, K, SAddr) to Req
where lookup(NAddr, Req, K) @ NAddr & node(NAddr, N)
& succ(NAddr, Succ, SAddr) & K in (N, Succ)

send lookup(Req, K, SAddr) to SAddr
where lookup(NAddr, Req, K) @ NAddr & node(NAddr, N)
& succ(NAddr, Succ, SAddr) & K not in (N, Succ)
From query to dataflow

send response( Req, K, SAddr ) to Req
where lookup( NAddr, Req, K ) @ NAddr & node( NAddr, N ) & succ( NAddr, Succ, SAddr ) & K in ( N, Succ )

send lookup( Req, K, SAddr ) to SAddr
where lookup( NAddr, Req, K ) @ NAddr & node( NAddr, N ) & succ( NAddr, Succ, SAddr ) & K not in ( N, Succ )
From query to dataflow

send response( Req, K, SAddr ) to Req
where lookup( NAddr, Req, K ) @ NAddr & node( NAddr, N )
& succ( NAddr, Succ, SAddr ) & K in ( N, Succ ]

send lookup( Req, K, SAddr ) to SAddr
where lookup( NAddr, Req, K ) @ NAddr & node( NAddr, N )
& succ( NAddr, Succ, SAddr ) & K not in ( N, Succ ]
From query to dataflow

send response( Req, K, SAddr ) to Req
where lookup( NAddr, Req, K ) @ NAddr & node( NAddr, N ) & succ( NAddr, Succ, SAddr ) & K in ( N, Succ ]

send lookup( Req, K, SAddr ) to SAddr
where lookup( NAddr, Req, K ) @ Naddr & node( NAddr, N )
& succ( NAddr, Succ, SAddr ) & K not in ( N, Succ ]

Diagram:

- Lookup
  - Join lookup.NI == node.NI
- Node
- Successor
  - Join lookup.NI == succ.NI
- Select K in (N, S]
- Project response@R (R, K, SI)
- Select K not in (N, S]
- NI, R, K, N, S, SI
  - K in (S, N]
send response( Req, K, SAddr ) to Req
where lookup( NAddr, Req, K ) @ NAddr & node ( NAddr, N ) & succ ( NAddr, Succ, SAddr ) & K in ( N, Succ ]

send lookup( Req, K, SAddr ) to SAddr
where lookup( NAddr, Req, K ) @ Naddr & node ( NAddr, N ) & succ ( NAddr, Succ, SAddr ) & K not in ( N, Succ ]
From query to dataflow

- One *strand* per subquery
- Strand order is immaterial
- Strands could execute in parallel
From query to dataflow

Strand 1

Strand 2

node  succ  ...
1. Distributed system specified in a query language

3. Graph executed directly to maintain routing tables and network overlay state

2. Compiled into optimized graph of dataflow elements

Packets in → Strand 1 → Strand 2 → ... → Packets out

Network Overlay Description

Strand 1:
- Packets out
- Packets in
- Distributed system
- Compiled into optimized graph
- Graph executed

Strand 2:
- Packets out
- Packets in

Timothy Roscoe
**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
Implementation

- Extensive library of elements
  - Relational operators
  - Queues, buffers, schedulers
  - Transport stack (more later)

- C++ and Python/Tcl bindings
  - Allows graph specification as with Click

- But wait – there’s more…
Query language

- Based on Datalog:
  - Based on Prolog with no imperative constructs
  - Fairly standard query language from literature
- Goals:
  - Understand language issues
  - Limit constructs as little as possible
  - Demonstrate benefits of conciseness
- Non-goals (at this stage):
  - A nice language to write in (as we will see)
  - Clean semantics (though we now have some)
  - Truly high-level, global property specification
Datalog and location specifiers

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

- Node state tuples
  - \( \text{node}(\text{NAddr}, N) \)
  - \( \text{successor}(\text{NAddr}, \text{Succ}, \text{SAddr}) \)

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

R1 response\( @\text{Req}(\text{Req}, K, \text{SAddr}) :- \)

\[ \text{lookup@NAddr}(\text{NAddr}, \text{Req}, K), \]

\[ \text{node@NAddr}(\text{NAddr}, N), \]

\[ \text{succ@NAddr}(\text{NAddr}, \text{Succ}, \text{SAddr}), \]

\( K \) in \( (N, \text{Succ})]. \)

R2 lookup\( @\text{SAddr}(\text{SAddr}, \text{Req}, K) :- \)

\[ \text{lookup@NAddr}(\text{NAddr}, \text{Req}, K), \]

\[ \text{node@NAddr}(\text{NAddr}, N), \]

\[ \text{succ@NAddr}(\text{NAddr}, \text{Succ}, \text{SAddr}), \]

\( K \) not in \( (N, \text{Succ})]. \)
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 diameter & state
    • Consistent routing with churn
    • Property checks as additional queries
  • 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
Maintenance bandwidth
(comparable with MIT Chord)
Latency without churn

CDF

Latency (s)

100 nodes
300 nodes
500 nodes
Latency under churn

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

![Graph showing CDF for consistent fraction over time]
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) & Peter Druschel (MPI)

- 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!
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 are 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?

http://P2.berkeley.intel-research.net