Declarative Overlays

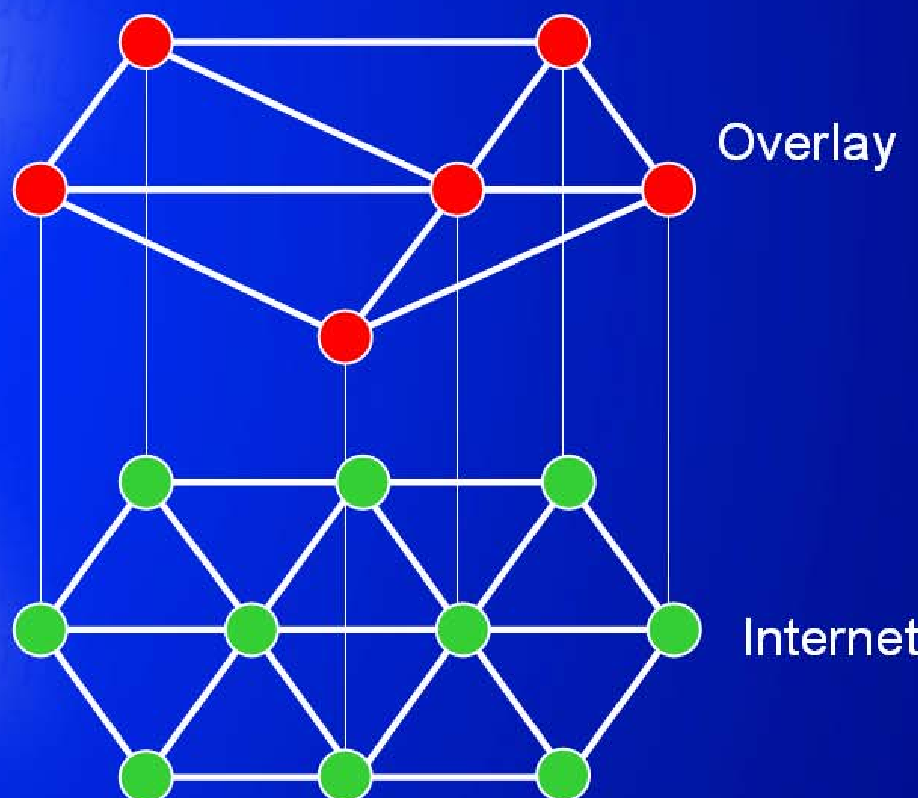
Petros Maniatis

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

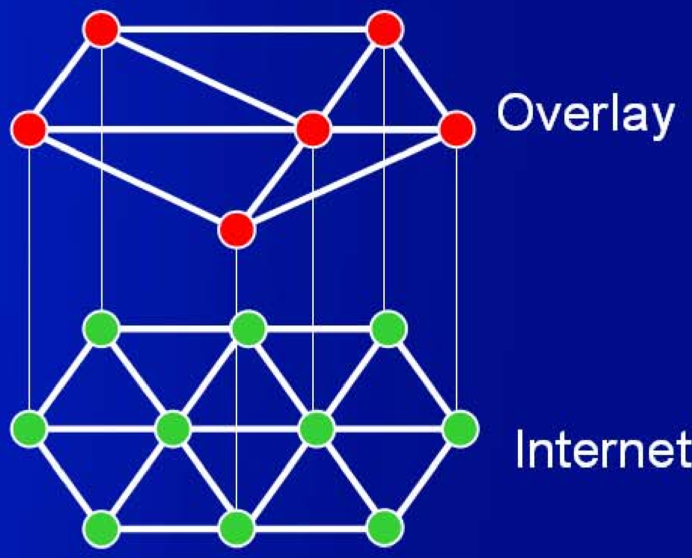
Overlays Everywhere…

"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

  - 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
  - Repeat
- But in practice
  - No global view
  - Wrong choice of algorithms
  - Homicidal boredom

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

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

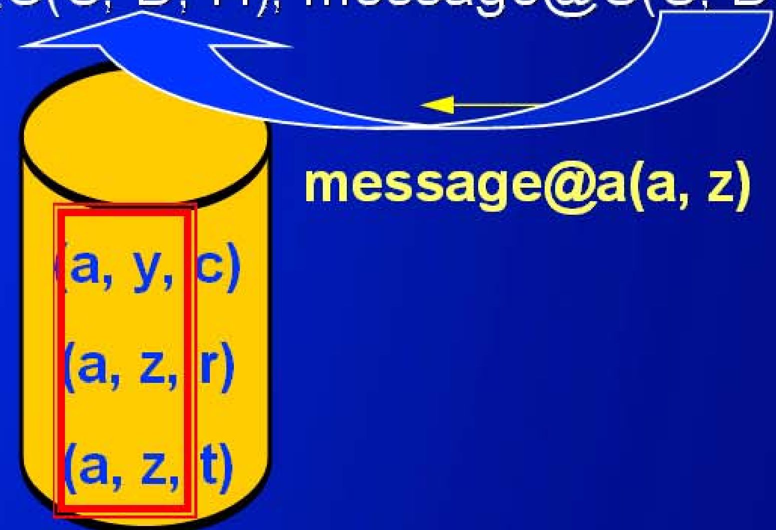
- Distributed state
  - Distributed soft state in relational tables, holding tuples of values
    - route (S, D, H)
  - Non-stored information passes around as *event tuple streams*
    - message (X, D)
- Overlay specification in declarative logic language (OverLog)
  - <head> :- <precondition1>, <precondition2>, … , <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

- 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 @Loc *place* individual tuples at specific nodes
  - message@H(H, D) :- route@S(S, D, H), message@S(S, D).

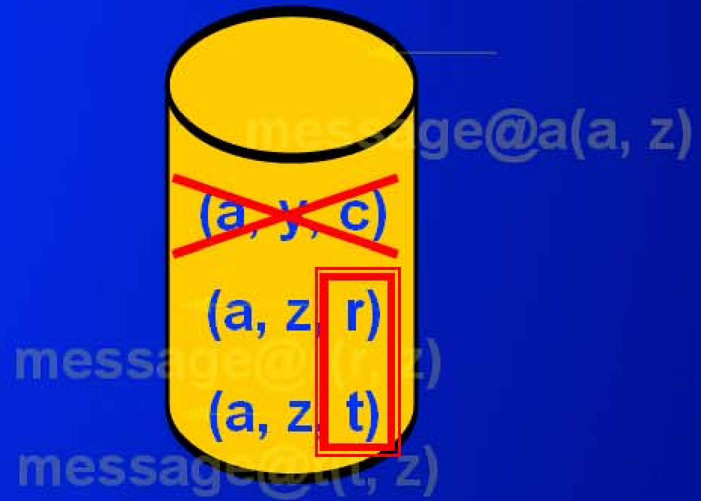
Enter P2: Semantics

- 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).



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).

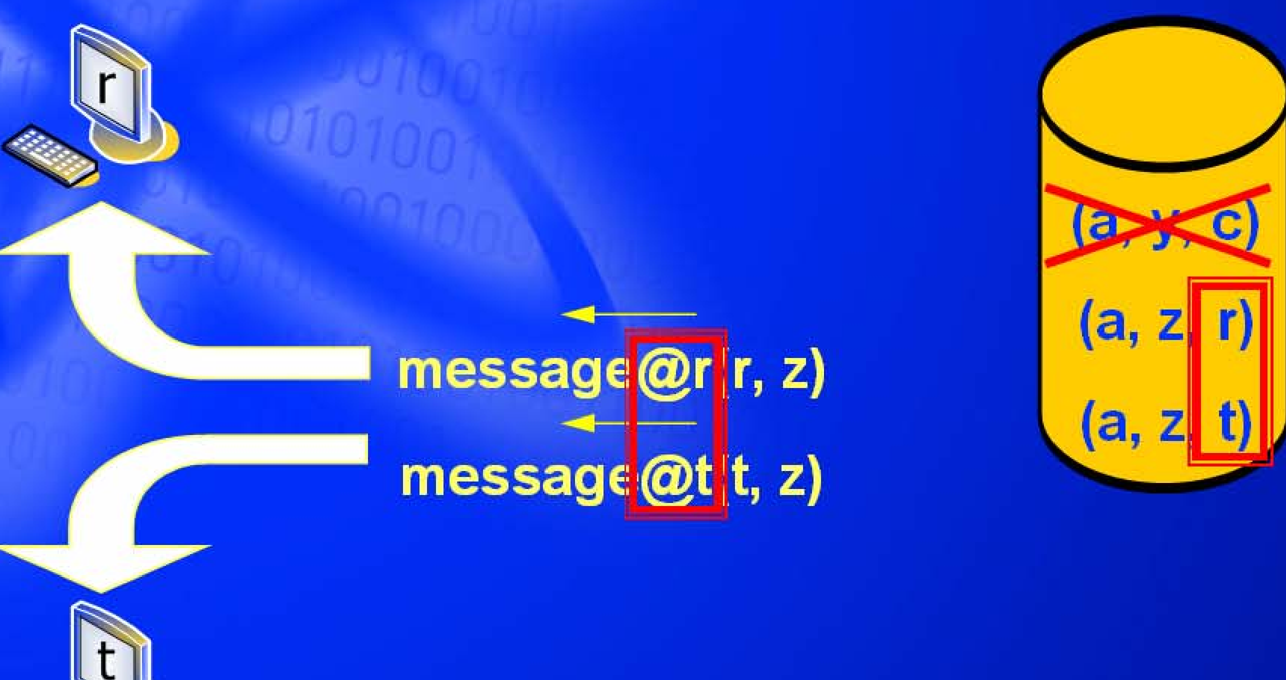


Enter P2: Semantics

- 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).

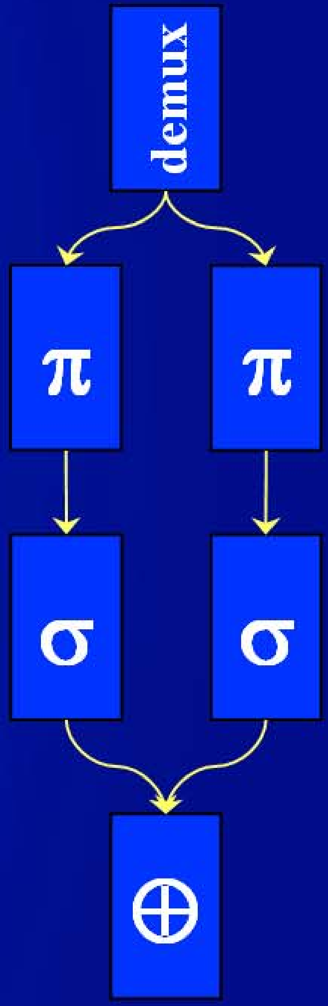
Enter P2: Semantics

- 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: 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

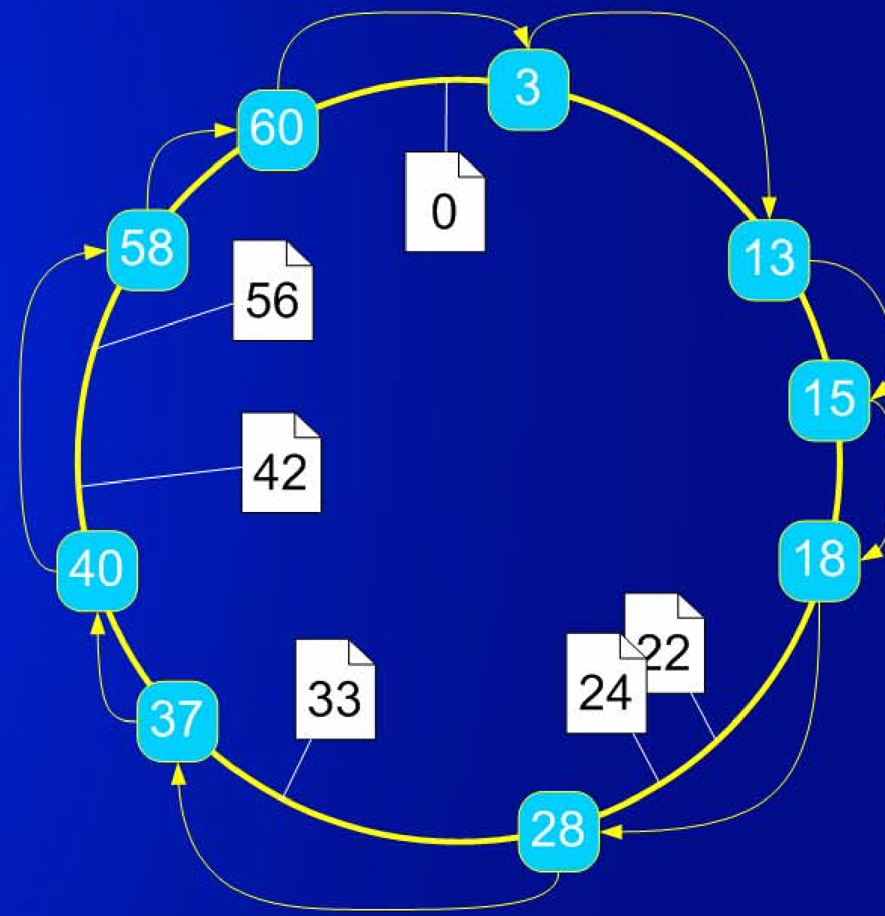
- Specification automatically translated to a dataflow graph
  - C++ dataflow elements (akin to Click elements)
  - Implement
    - relational operators (joins, selections, projections)

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



P2

Example: Ring Routing

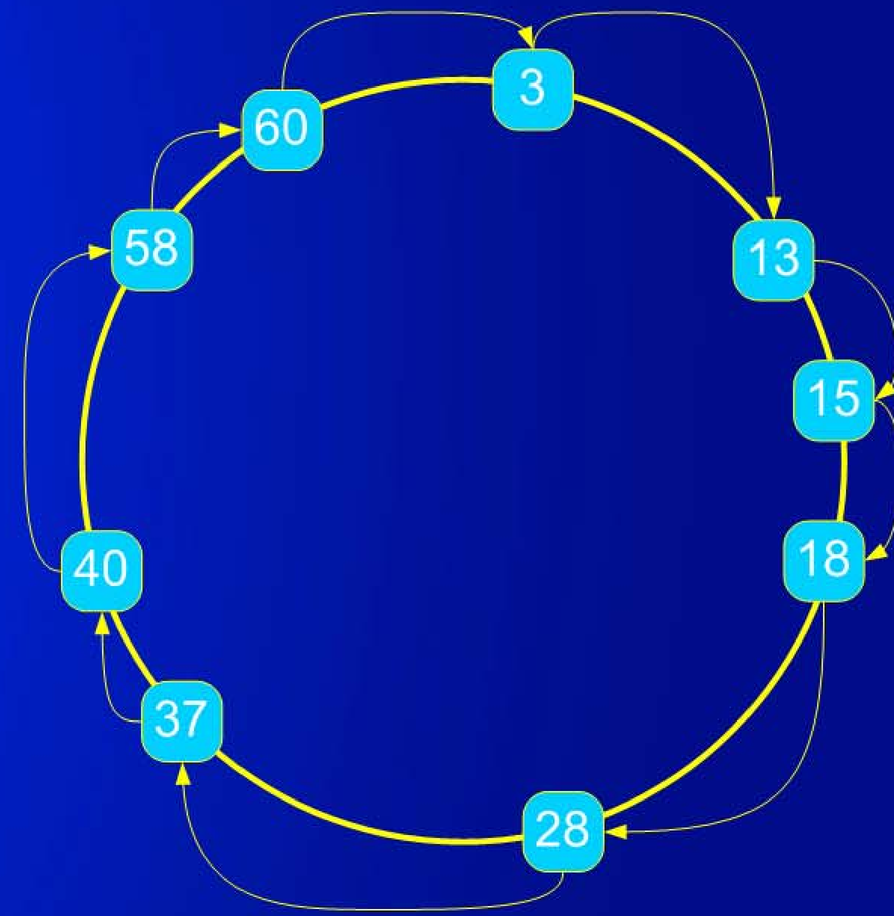
- How do I find the responsible node for a given key k?
- <u>n.lookup(k)</u>

  if k in (n, n.successor)

  return n.successor

  else

  return n.successor. lookup(k)



Ring State

- n.lookup(k)

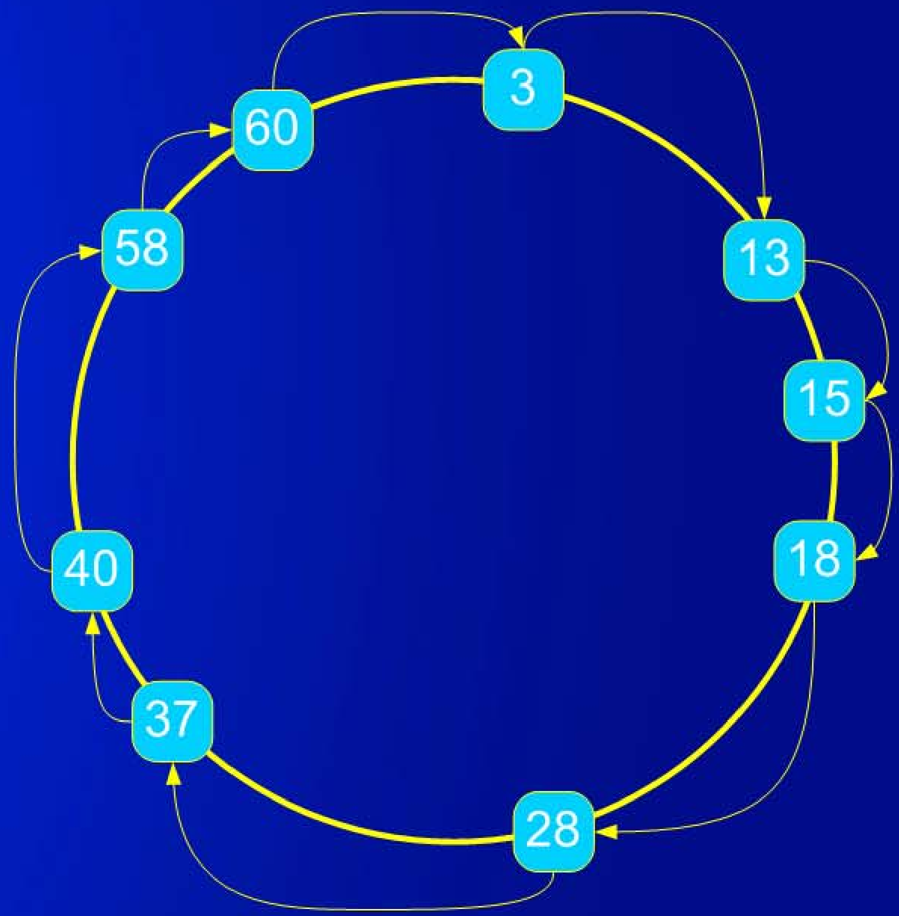
  if k in (n, n.successor)

  return n.successor

  else

  return n.successor. lookup(k)

- Node state tuples
  - node(NAddr, N)
  - successor(NAddr, Succ, SAddr)
- Transient event tuples
  - lookup (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(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].
```

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.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(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].

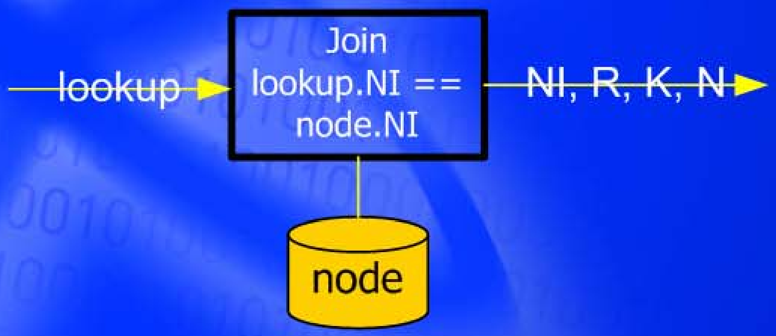
R2 lookup@SI(SI, R, K) :- lookup@NI(NI, R, K),
node@NI(NI, N), succ@NI(NI, S, SI), K not in (N, S].

lookup

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].

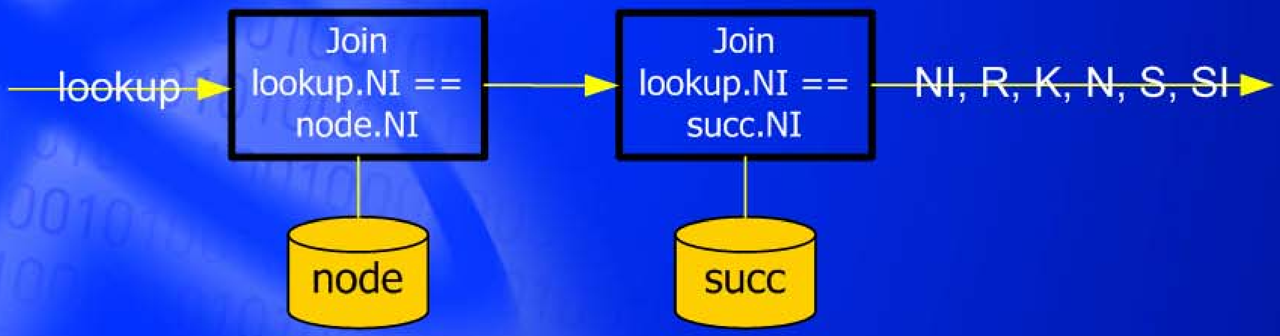
R2 lookup@SI(SI, R, K) :- lookup@NI(NI, R, K), node@NI(NI, N), succ@NI(NI, S, SI), K not in (N, S].



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].

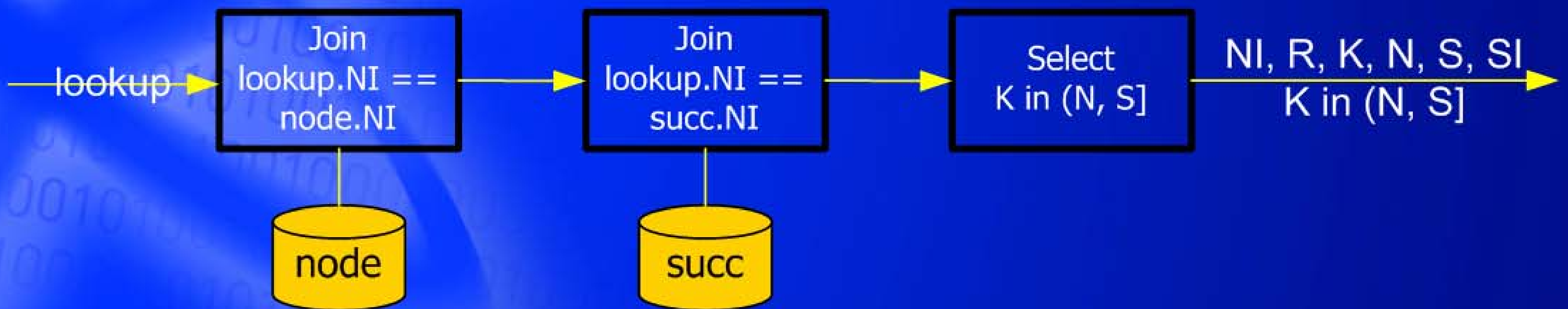
R2 lookup@SI(SI, R, K) :- lookup@NI(NI, R, K), node@NI(NI, N), succ@NI(NI, S, SI), K not in (N, S].



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]**.

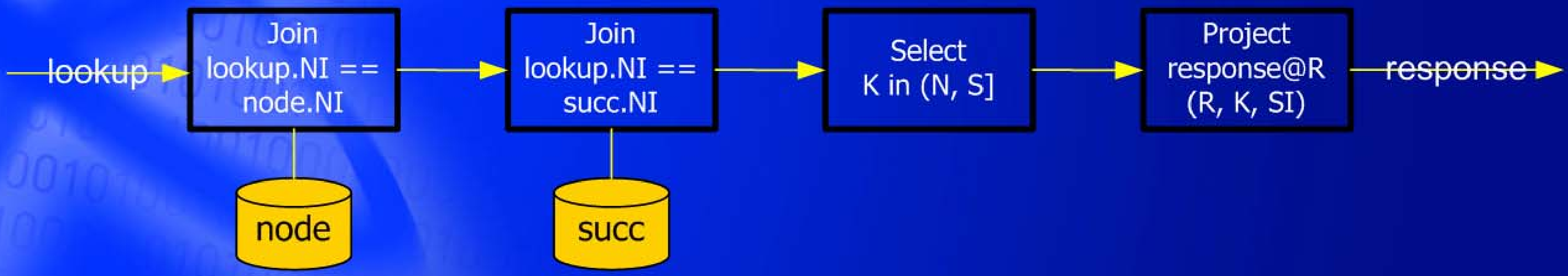
R2 lookup@SI(SI, R, K) :- lookup@NI(NI, R, K), node@NI(NI, N), succ@NI(NI, S, SI), K not in (N, S].



From OverLog to Dataflow

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

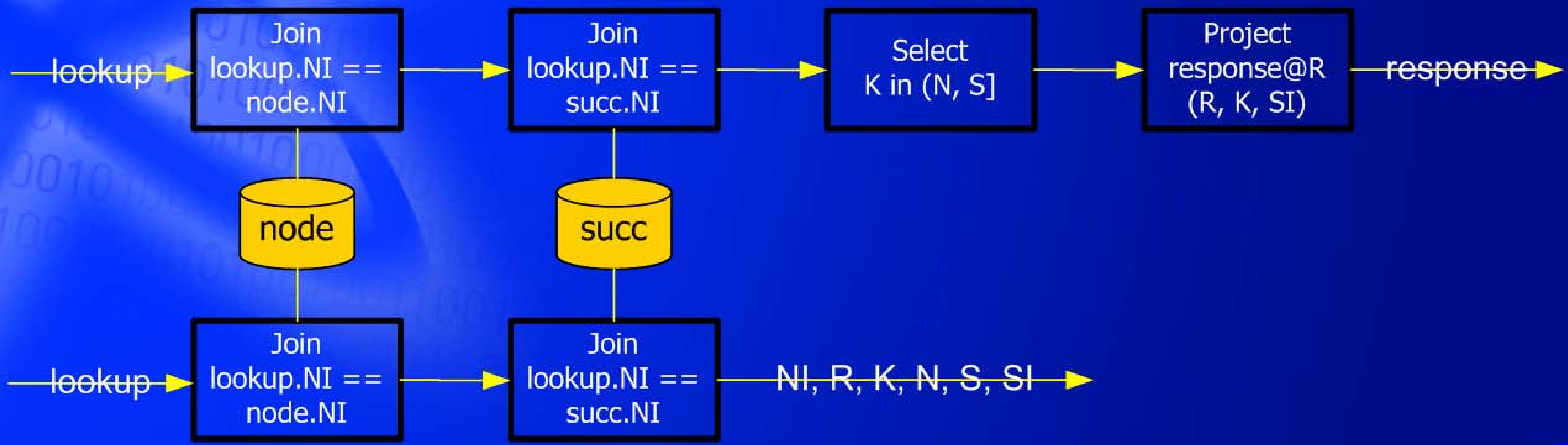
R2 lookup@SI(SI, R, K) :- lookup@NI(NI, R, K),
node@NI(NI, N), succ@NI(NI, S, SI), K not in (N, S].



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].

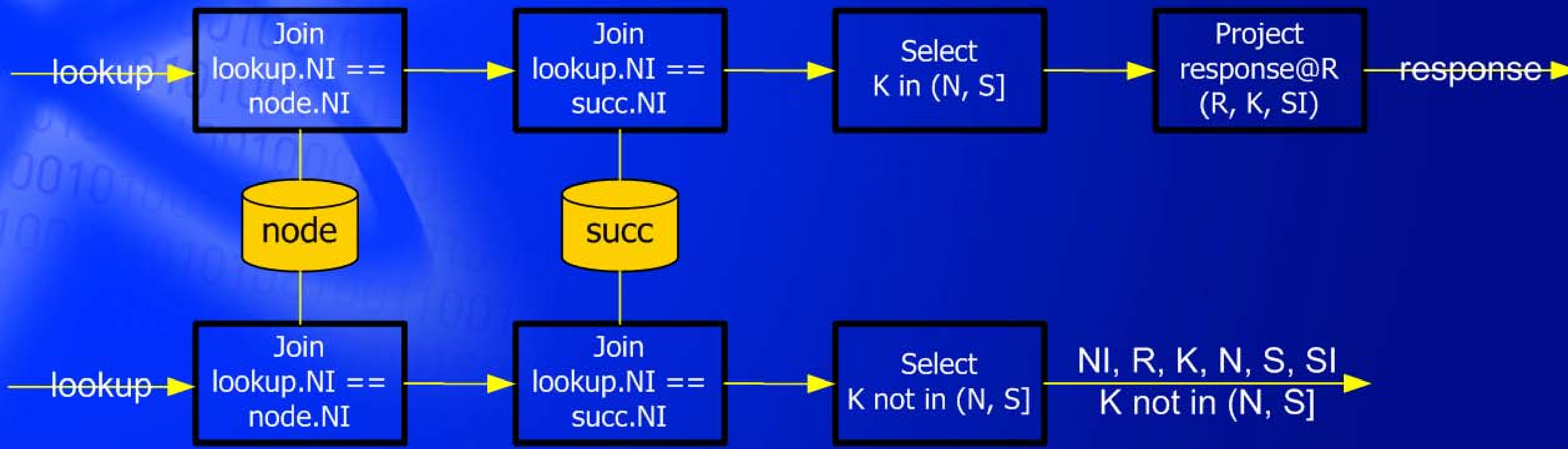
R2 lookup@SI(SI, R, K) :- **lookup@NI(NI, R, K), node@NI(NI, N), succ@NI(NI, S, SI),** K not in (N, S].



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].

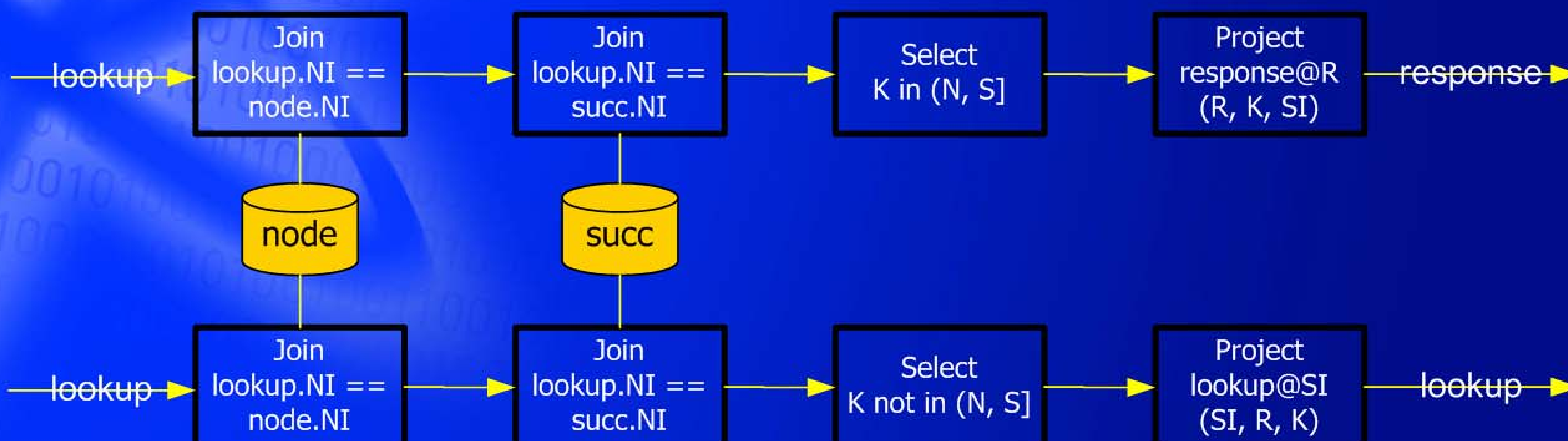
R2 lookup@SI(SI, R, K) :- **lookup@NI(NI, R, K),
node@NI(NI, N), succ@NI(NI, S, SI), K not in (N, S]**.



From OverLog to Dataflow

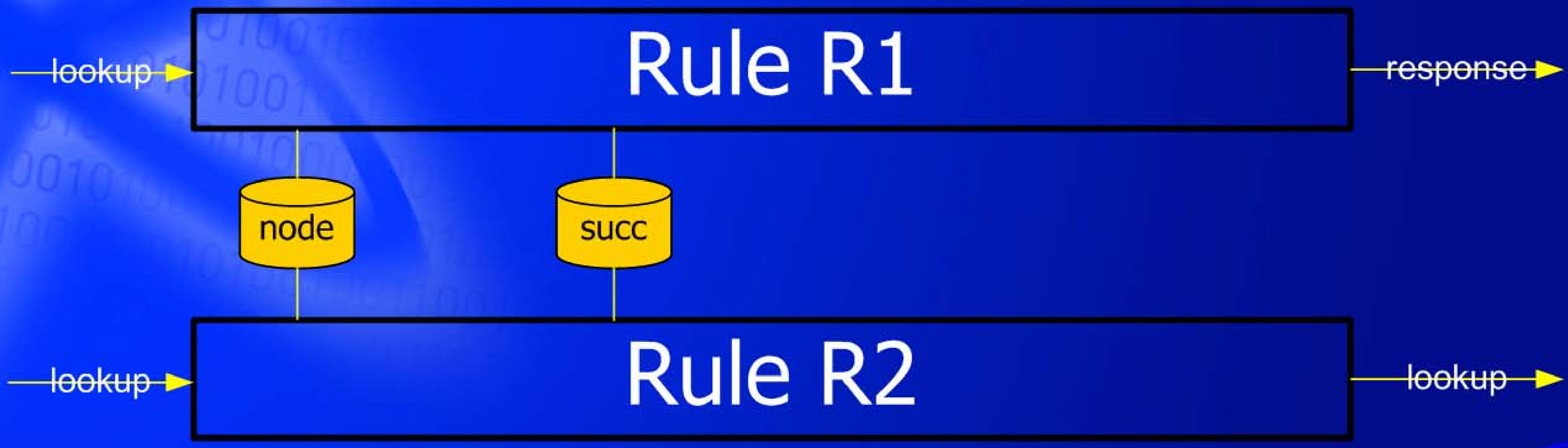
R1 response@R(R, K, SI) : - lookup@NI(NI, R, K),
node@NI(NI, N), succ@NI(NI, S, SI), K in (N, S].

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].

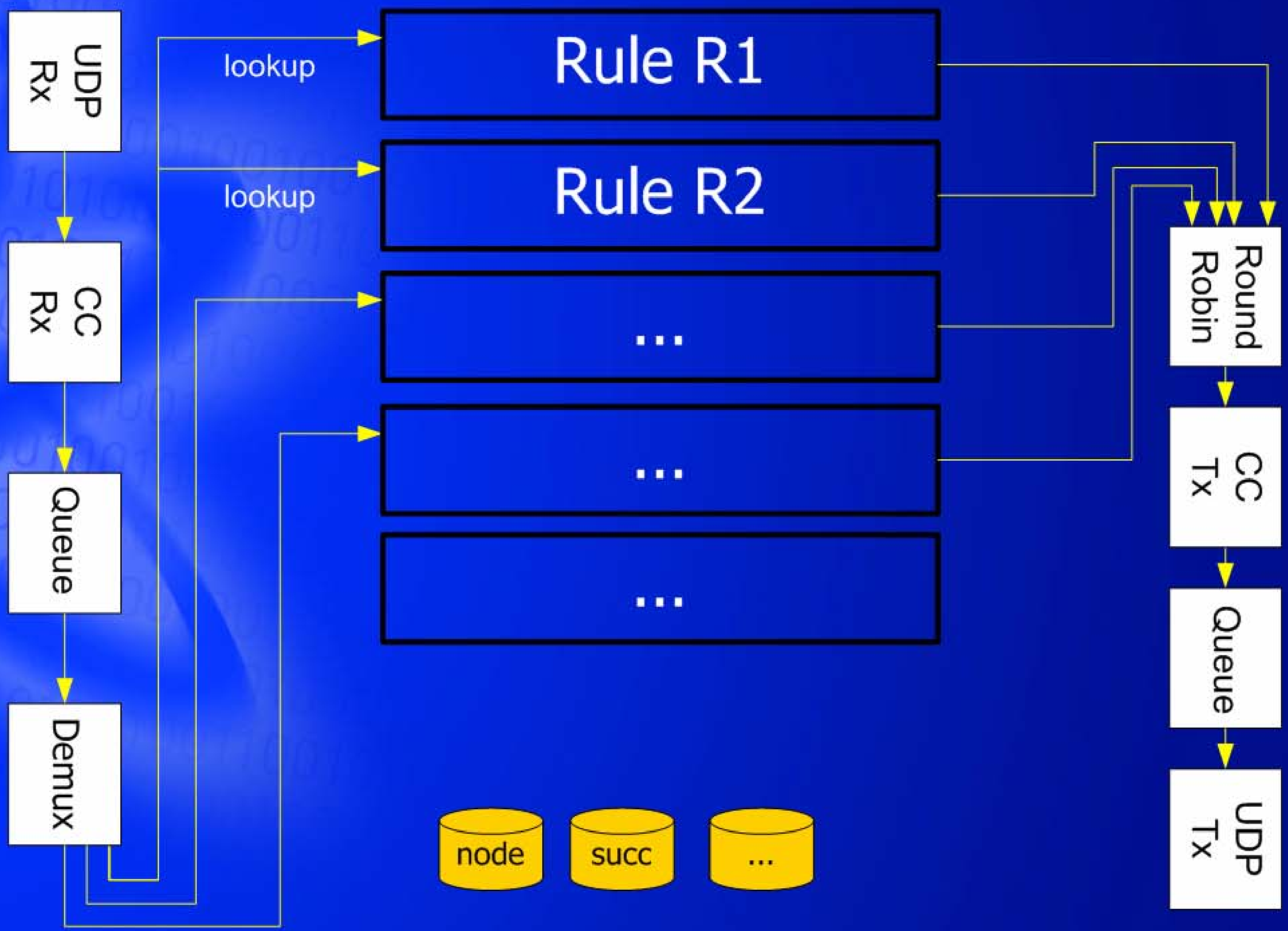


From OverLog to Dataflow

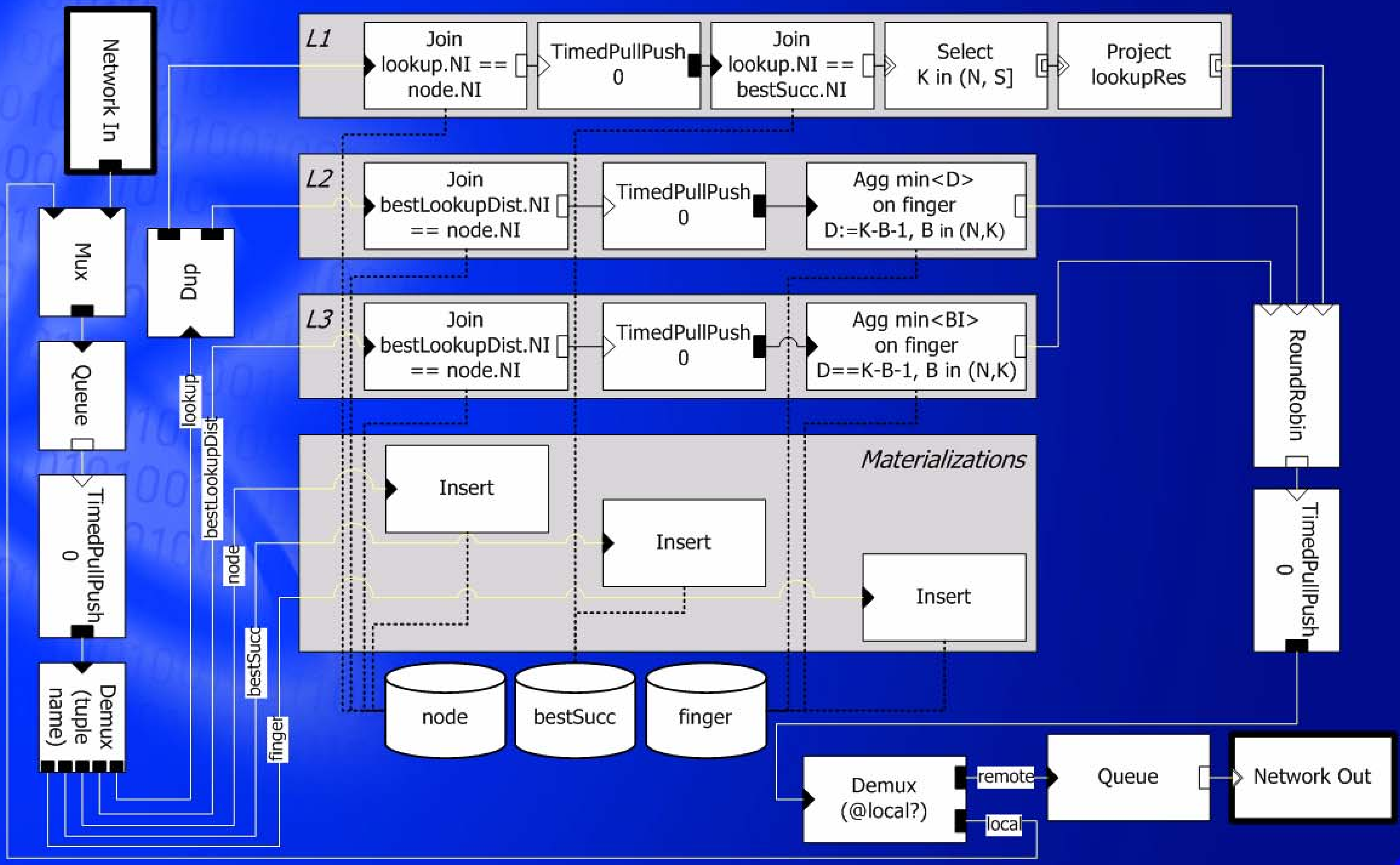
- One *rule strand* per OverLog rule
- Rule order is immaterial
- Rule strands could execute in parallel



Transport and App Logic



A Bit of Chord



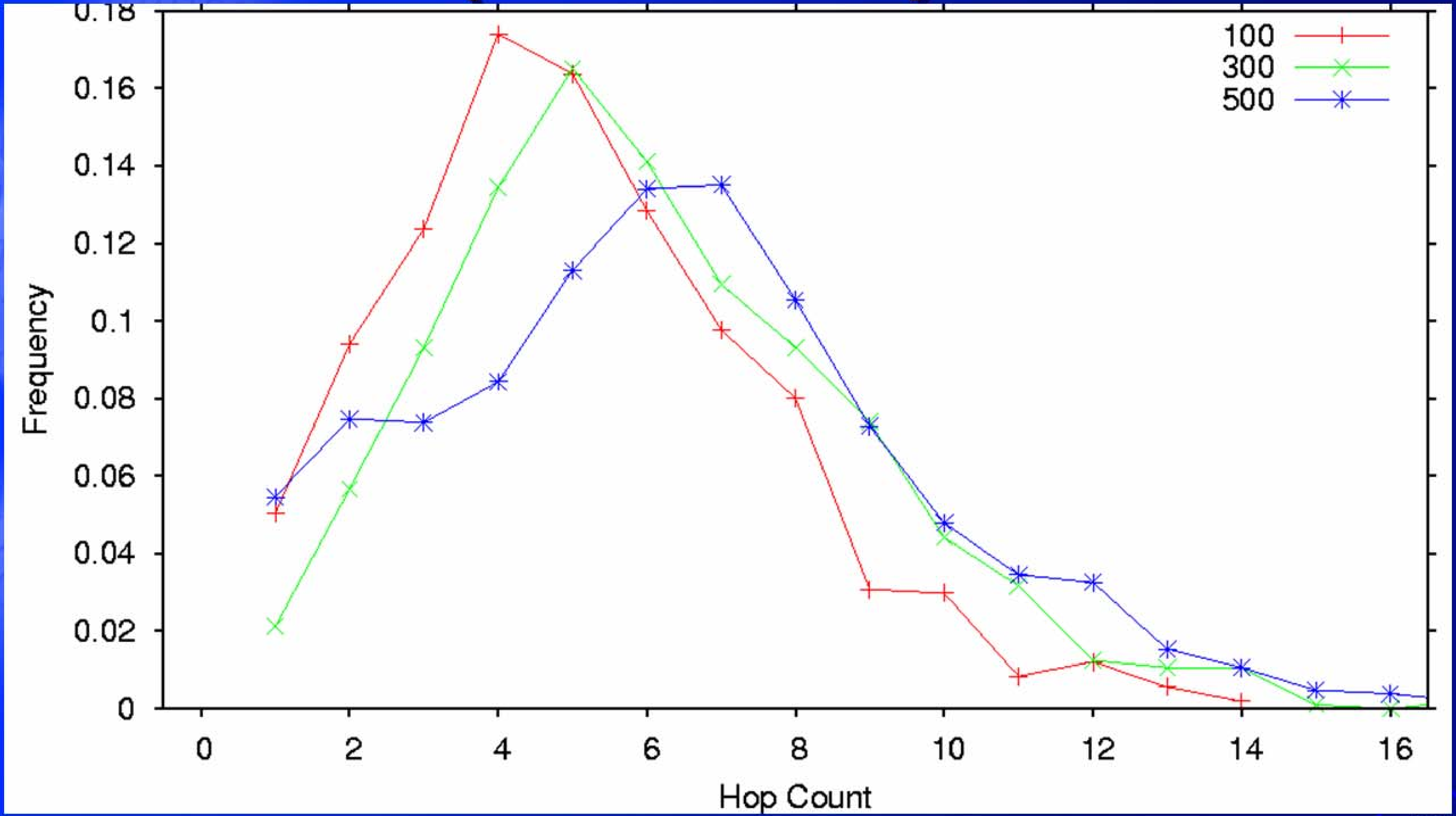
Chord on P2

- 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

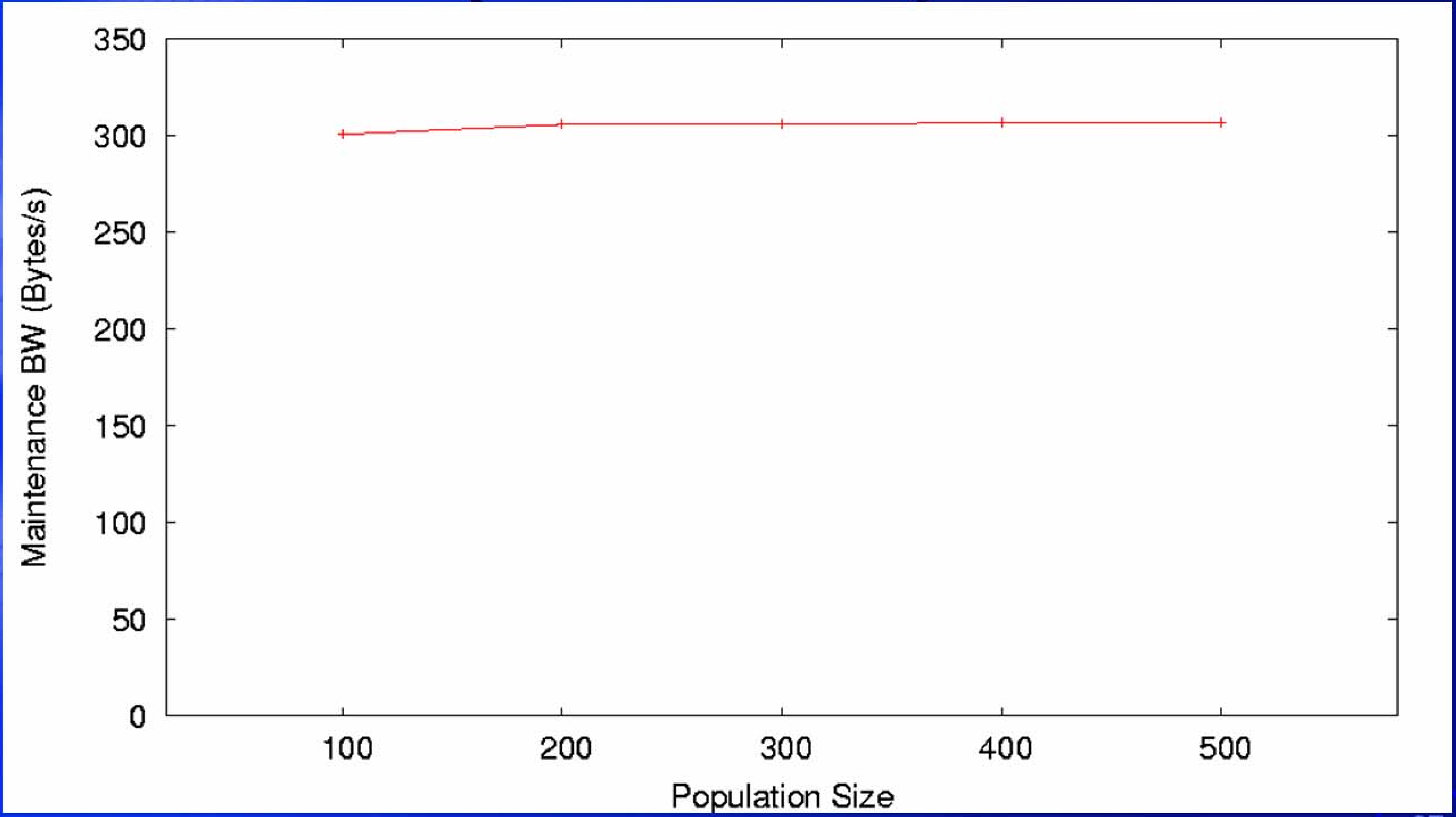
and node failure detection.

```
/* The base tuples */

materialize(node, infinity, 1, keys(1)).
materialize(finger, 180, 160, keys(2)).
materialize(bestSucc, infinity, 1, keys(1)).
materialize(succDist, 10, 100, keys(2)).
materialize(succ, 10, 100, keys(2)).
materialize(pred, infinity, 100, keys(1)).
materialize(succCount, infinity, 1, keys(1)).
materialize(join, 10, 5, keys(1)).
materialize(landmark, infinity, 1, keys(1)).
materialize(fFix, infinity, 160, keys(2)).
materialize(nextFingerFix, infinity, 1, keys(1)).
materialize(pingNode, 10, infinity, keys(2)).
materialize(pendingPing, 10, infinity, keys(2)).

/** Lookups */

L1 lookupResults@R(R,K,S,SI,E) :- node@NI(NI,N),
  lookup@NI(NI,K,R,E), bestSucc@NI(NI,S,SI), K in
  (N,S].
L2 bestLookupDist@NI(NI,K,R,E,min<D>) :-
  node@NI(NI,N), lookup@NI(NI,K,R,E),
  finger@NI(NI,I,B,BI), D:=K - B - 1, B in (N,K).
L3 lookup@BI(min<BI>,K,R,E) :- node@NI(NI,N),
  bestLookupDist@NI(NI,K,R,E,D),
  finger@NI(NI,I,B,BI), D == K - B - 1, B in (N,K).

/** Neighbor Selection */

N1 succEvent@NI(NI,S,SI) :- succ@NI(NI,S,SI).
N2 succDist@NI(NI,S,D) :- node@NI(NI,N),
  succEvent@NI(NI,S,SI), D:=S - N - 1.
N3 bestSuccDist@NI(NI,min<D>) :-
  succDist@NI(NI,S,D).
N4 bestSucc@NI(NI,S,SI) :- succ@NI(NI,S,SI),
  bestSuccDist@NI(NI,D), node@NI(NI,N), D == S - N
  - 1.
N5 finger@NI(NI,0,S,SI) :- bestSucc@NI(NI,S,SI).

/** Successor eviction */

S1 succCount@NI(NI,count<*>) :- succ@NI(NI,S,SI).
S2 evictSucc@NI(NI) :- succCount@NI(NI,C), C > 4.
S3 maxSuccDist@NI(NI,max<D>) :- succ@NI(NI,S,SI),
  node@NI(NI,N), evictSucc@NI(NI), D:=S - N - 1.
S4 delete succ@NI(NI,S,SI) :- node@NI(NI,N),
  succ@NI(NI,S,SI), maxSuccDist@NI(NI,D), D == S -
  N - 1.

/** Finger fixing */

F0 nextFingerFix@NI(NI, 0).
F1 fFix@NI(NI,E,I) :- periodic@NI(NI,E,10),
  nextFingerFix@NI(NI,I).
F2 fFixEvent@NI(NI,E,I) :- fFix@NI(NI,E,I).
F3 lookup@NI(NI,K,NI,E) :- fFixEvent@NI(NI,E,I),
  node@NI(NI,N), K:=1I << I + N.
F4 eagerFinger@NI(NI,I,B,BI) :- fFix@NI(NI,E,I),
  lookupResults@NI(NI,K,B,BI,E).
F5 finger@NI(NI,I,B,BI) :-
  eagerFinger@NI(NI,I,B,BI).
F6 eagerFinger@NI(NI,I,B,BI) :- node@NI(NI,N),
  eagerFinger@NI(NI,I1,B,BI), I:=I1 + 1, K:=1I << I
  + N, K in (N,B), BI != NI.
F7 delete fFix@NI(NI,E,I1) :-
  eagerFinger@NI(NI,I,B,BI), fFix@NI(NI,E,I1), I >
  0, I1 == I - 1.
F8 nextFingerFix@NI(NI,0) :-
  eagerFinger@NI(NI,I,B,BI), ((I == 159) || (BI ==
  NI)).
F9 nextFingerFix@NI(NI,I) :- node@NI(NI,N),
  eagerFinger@NI(NI,I1,B,BI), I:=I1 + 1, K:=1I << I
  + N, K in (B,N), NI != BI.

/** Churn Handling */

C1 joinEvent@NI(NI,E) :- join@NI(NI,E).
C2 joinReq@LI(LI,N,NI,E) :- joinEvent@NI(NI,E),
  node@NI(NI,N), landmark@NI(NI,LI), LI != "-".
C3 succ@NI(NI,N,NI) :- landmark@NI(NI,LI),
  joinEvent@NI(NI,E), node@NI(NI,N), LI == "-".
C4 lookup@LI(LI,N,NI,E) :- joinReq@LI(LI,N,NI,E).
C5 succ@NI(NI,S,SI) :- join@NI(NI,E),
  lookupResults@NI(NI,K,S,SI,E).

/** Stabilization */

SB0 pred@NI(NI, P) :- P := "-".
SB1 stabilize@NI(NI,E) :- periodic@NI(NI,E,15).
SB2 stabilizeRequest@SI(SI,NI) :-
  stabilize@NI(NI,E), bestSucc@NI(NI,S,SI).
SB3 sendPredecessor@PI1(PI1,P,PI) :-
  stabilizeRequest@NI(NI,PI1), pred@NI(NI,P,PI), PI
  != "-".
SB4 succ@NI(NI,P,PI) :- node@NI(NI,N),
  sendPredecessor@NI(NI,P,PI),
  bestSucc@NI(NI,S,SI), P in (N,S).
SB5 sendSuccessors@SI(SI,NI) :- stabilize@NI(NI,E),
  succ@NI(NI,S,SI).
SB6 returnSuccessor@PI(PI,S,SI) :-
  sendSuccessors@NI(NI,PI), succ@NI(NI,S,SI).
SB7 succ@NI(NI,S,SI) :-
  returnSuccessor@NI(NI,S,SI).
SB7 notifyPredecessor@SI(SI,N,NI) :-
  stabilize@NI(NI,E), node@NI(NI,N),
  succ@NI(NI,S,SI).
SB8 pred@NI(NI,P,PI) :- node@NI(NI,N),
  notifyPredecessor@NI(NI,P,PI),
  pred@NI(NI,P1,PI1), ((PI1 == "-") || (P in
  (P1,N))).

/** Connectivity Monitoring */

CM0 pingEvent@NI(NI,E) :- periodic@NI(NI,E,5).
CM1 pendingPing@NI(NI,PI,E) :- pingEvent@NI(NI,E),
  pingNode@NI(NI,PI).
CM2 pingReq@PI(PI,NI,E) :- pendingPing@NI(NI,PI,E).
CM3 delete pendingPing@NI(NI,PI,E) :-
  pingResp@NI(NI,PI,E).
CM4 pingResp@RI(RI,NI,E) :- pingReq@NI(NI,RI,E).
CM5 pingNode@NI(NI,SI) :- succ@NI(NI,S,SI), SI !=
  NI.
CM6 pingNode@NI(NI,PI) :- pred@NI(NI,P,PI), PI !=
  NI, PI != "-".
CM7 succ@NI(NI,S,SI) :- succ@NI(NI,S,SI),
  pingResp@NI(NI,SI,E).
CM8 pred@NI(NI,P,PI) :- pred@NI(NI,P,PI),
  pingResp@NI(NI,PI,E).
CM9 pred@NI(NI,"-","-") :- pingEvent@NI(NI,E),
  pendingPing@NI(NI,PI,E), pred@NI(NI,P,PI).
```

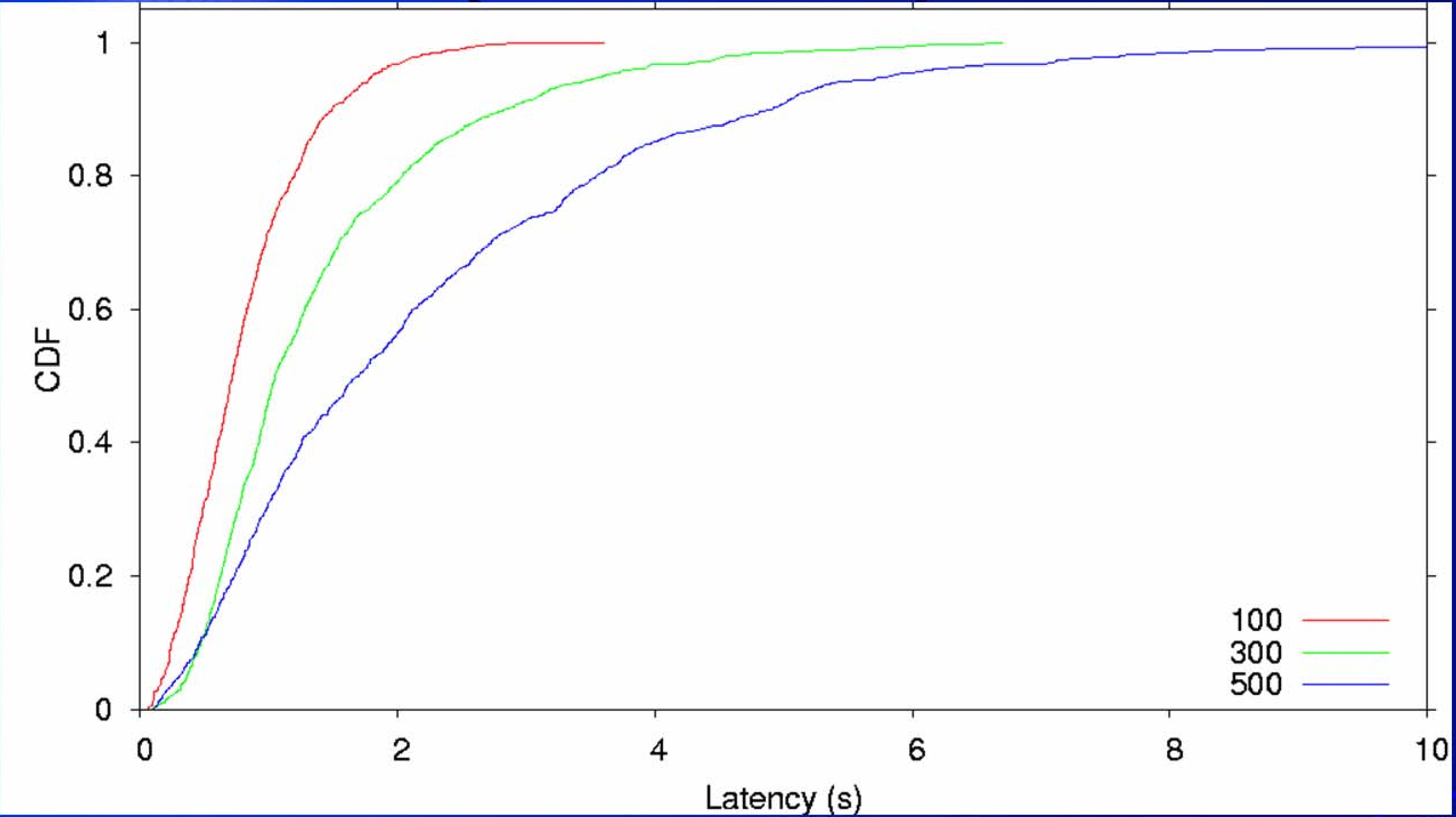
Lookup length in hops (no churn)



Maintenance bandwidth (no churn)

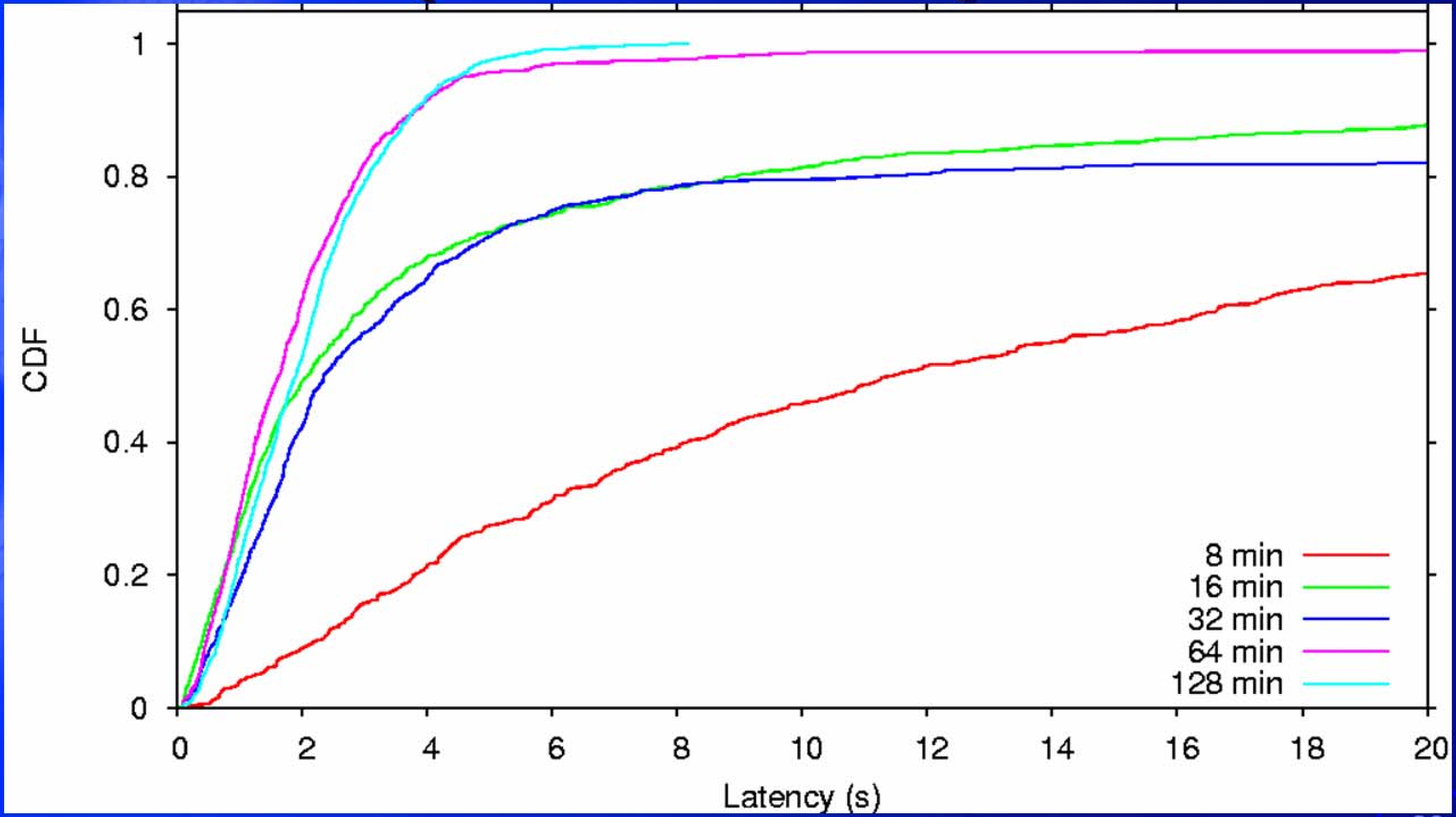


Lookup Latency (no churn)



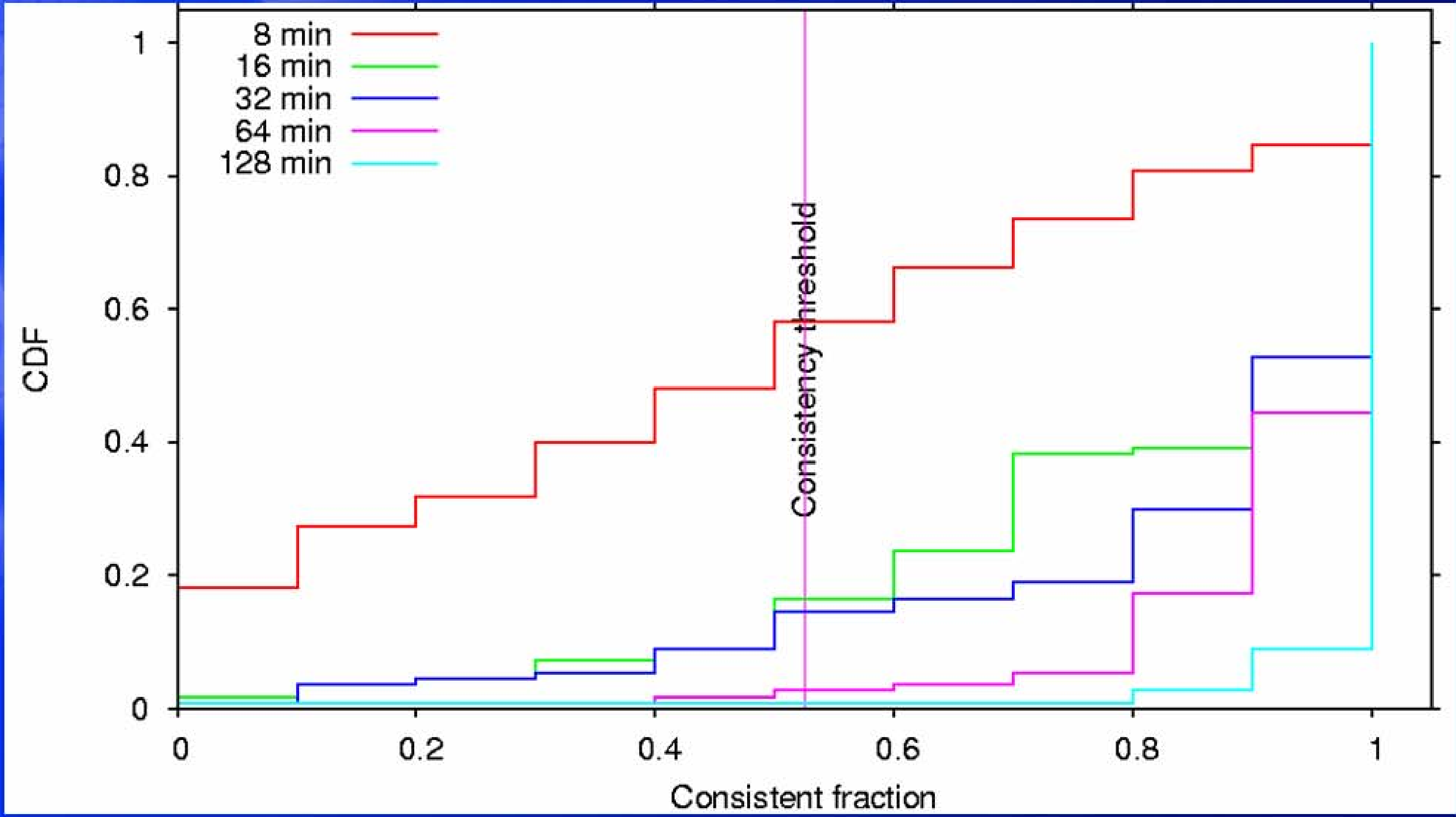
P2

Lookup Latency (with churn)

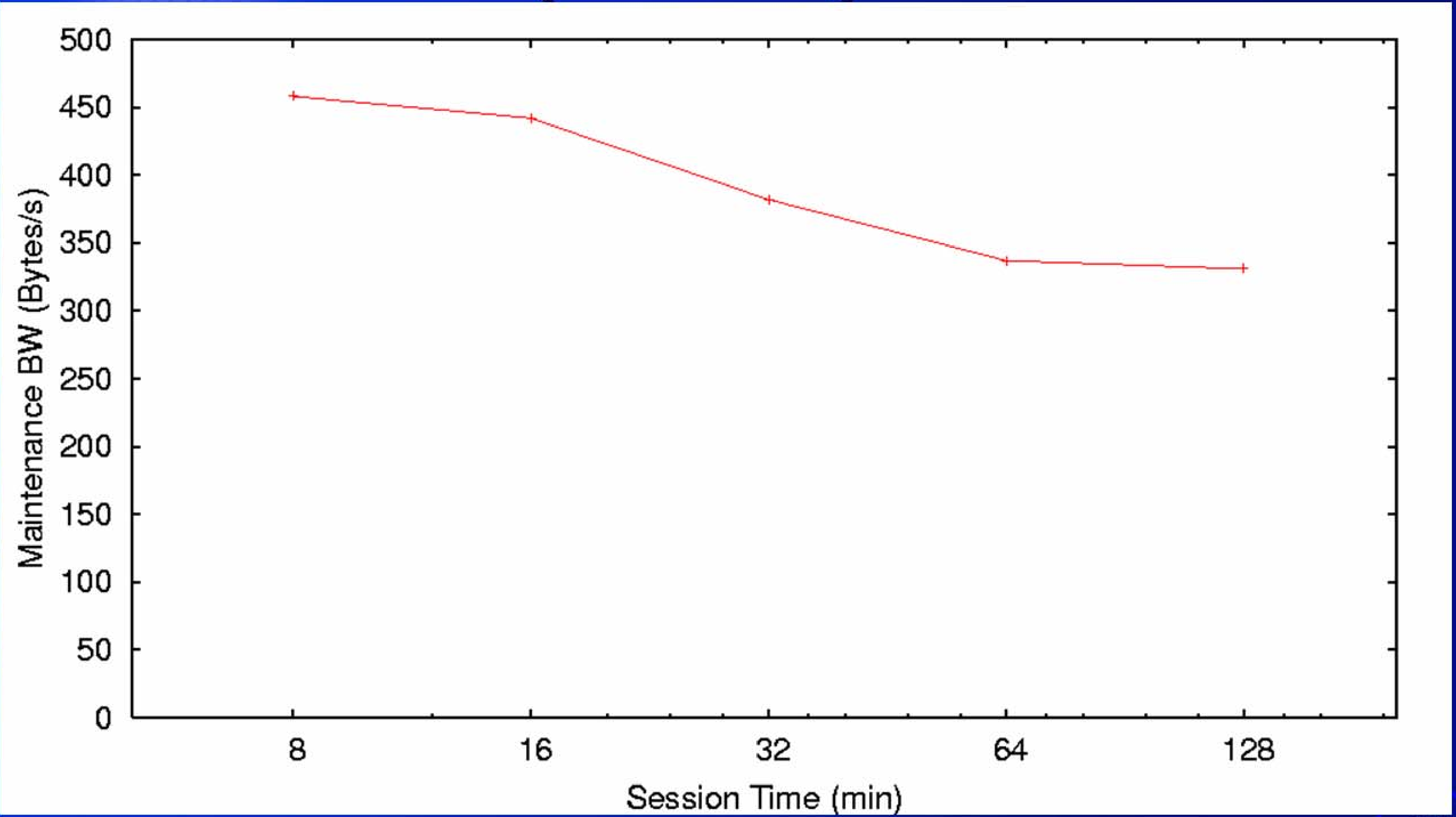


Lookup Consistency (with churn)

- 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

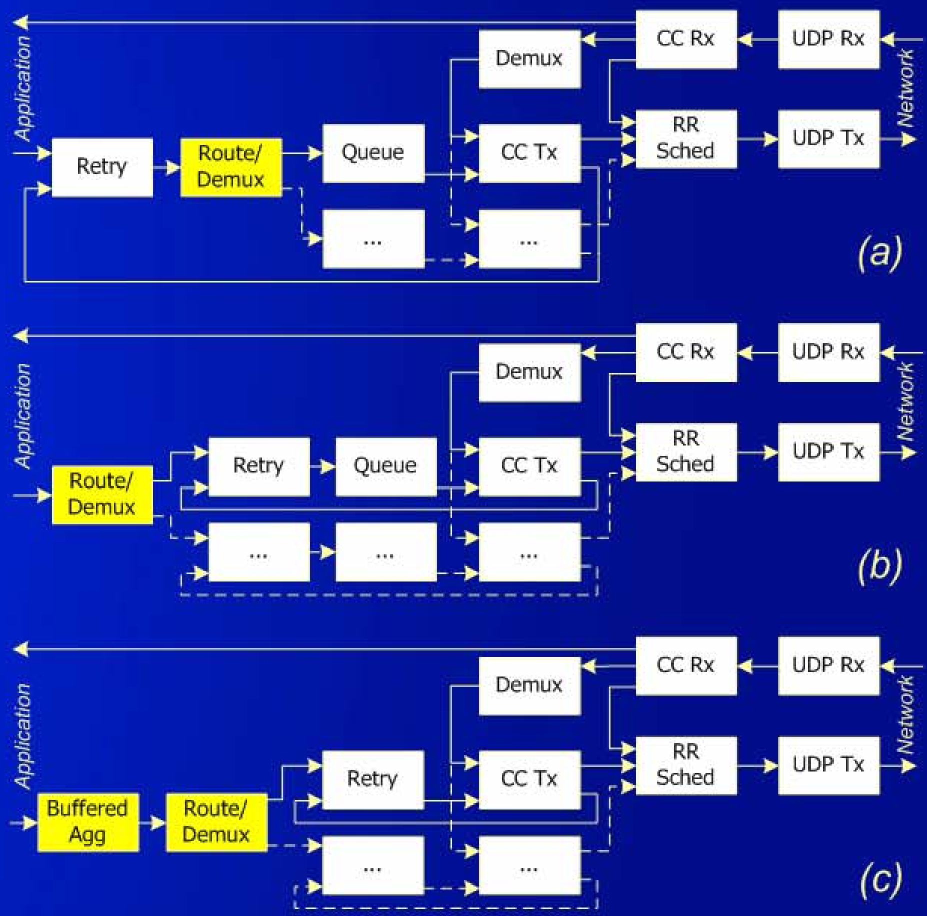
- Execution tracing/logging happens externally to system specification
  - At "pseudo-code" granularity: *logical stepping*
    - Why did rule R7 trigger? Under what preconditions?
    - Every rule execution (input and outputs) is exported as a table
      - ruleExec(Rule, InTuple, OutTuple, OutNode, 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?"

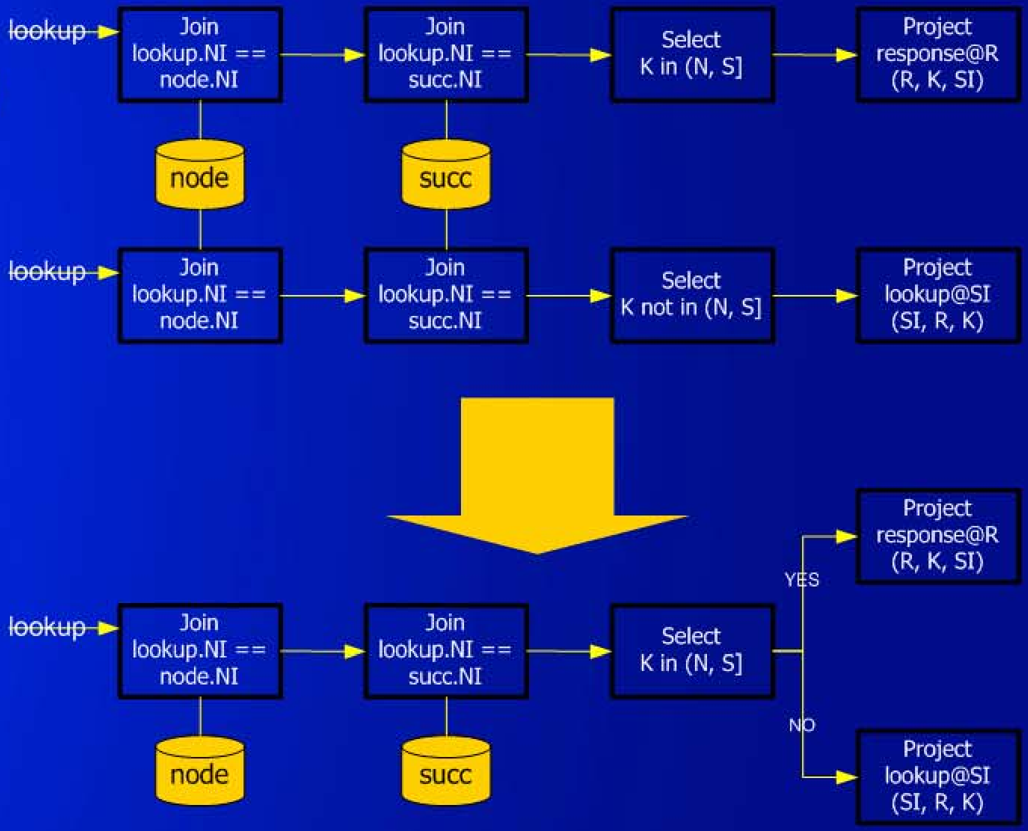
2. Reconfigurable Transport

- 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?