Implementing Declarative Overlays

Timothy Roscoe

Joint work with Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Ion Stoica

Intel Research and U.C. Berkeley

Tuesday, July 12, 2005

Broad Challenge: Network Routing Implementation

- Protocol-centric approach is usual:
  - Finite state automata
  - Asynchronous messages / events
  - Intuitive, but:
- Hard to:
  - reason about structure
  - check/debug
  - compose/abstract/reuse
- But few, if any, new abstractions have emerged for the problem.

Talk Overview

- Approach: take high-level view
  - Routing and Query Processing
  - Declarative specifications
- P2: a declarative overlay engine
  - OverLog language
  - Software dataflow implementation
- Evaluation: Chord as a test case
- Ongoing and future work

Declarative Networking

- 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 and 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

Distributed Continuous Query Processing

- Relatively new and active field
  - SDIMS, Mercury, IrisLog, Sophia, etc., in particular PIER
  - $\Rightarrow$ May not have all the answers yet
- But brings a wealth of experience and knowledge from database systems
  - Relational, deductive, stream processing, etc.

Goal: Declarative Networks

1. Express network properties as queries in a high-level declarative language
   - More than configuration or policy language
   - Apply static checking
   - Modular decomposition
2. Compile/interpret to maintain network
   - Dynamic optimization (e.g. eddies)
   - Sharing of computation/communication

Other advantages

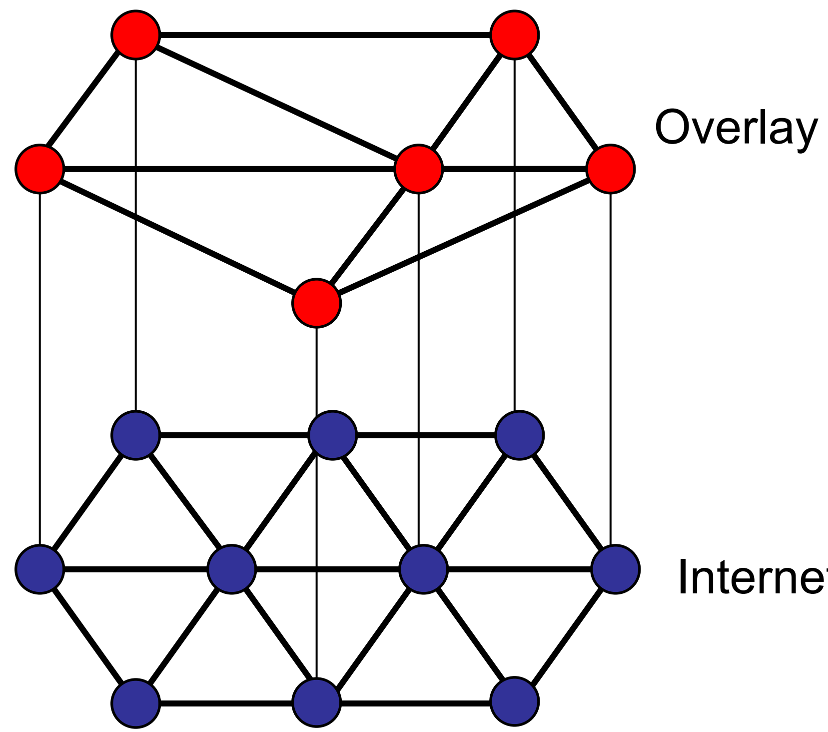
- Can incorporate other knowledge into routing policies
  - E.g., physical network knowledge
- Naturally integrates *discovery*
  - Often missing from current protocols
- Also provides an abstraction point for such information
  - Knowledge itself doesn't need to be exposed.

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

Specfic case: overlays

- Application level:
  - e.g. DHTs, P2P networks, ESM, etc.
- IP-oriented:
  - e.g. RON, IPVPNs, SOS, M/cast, etc.
- More generally: routing fn of any large distributed system
  - e.g MS Exchange, mgmt systems



Intel Research Berkeley

Why overlays?

- Overlays in a very *broad* sense
  - Any application-level routing system
  - Email servers, multicast, CDNs, DHTs, etc.
  - $\Rightarrow$ broad applicability
- Ideal test case
  - Clearly deployable short-term
  - Defers interoperability issues
  - Testbed for other domains
- The overlay design space is wide
  - $\Rightarrow$ ensure we cover the bases

Background

- PIER: distributed relational query processor (Huebsch et.al.)
  - Used DHT for hashing, trees, etc.
- Click: modular s/w forwarding engine (Kohler et.al.)
  - Used dataflow element graph
- XORP router (Handley et.al.)
  - Dataflow approach to BGP, OSPF, etc.

P2: A declarative overlay engine

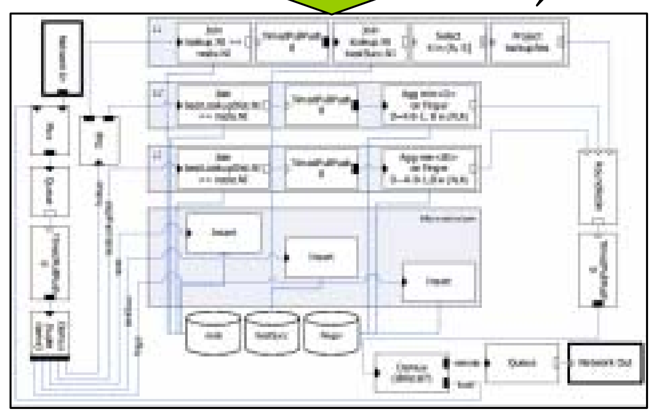
OverLog specification

Parser

Planner

Software Dataflow Graph

Received Packets



Sent Packets

Intel Research Berkeley

Data Model

- Relational tuples
- Two types of named relation:
  - Soft-state *tables*
  - *Streams* of transient tuples
- Simple, natural model for network state
  - Concisely expressed in a declarative language

Language: DataLog

- Well-known relational query language from the literature
  - Particularly deductive databases
  - Prolog with no imperative constructs
  - Equivalent to SQL with recursion
- *OverLog*: variant of DataLog
  - Streams & tables
  - Location specifiers for tuples

Why DataLog?

- Advantages:
  - Generality allows great flexibility
  - Easy to map prior optimization work
  - Simple syntax, easy to extend
- Disadvantages:
  - Hard for imperative programmers
  - Structure may not map to network concepts
- Good initial experimental vehicle

Overlog by example

- Gossiping a mesh:
  - materialise(neighbour, 1, 60, infinity).
  - materialise(member, 1, 60, infinity).

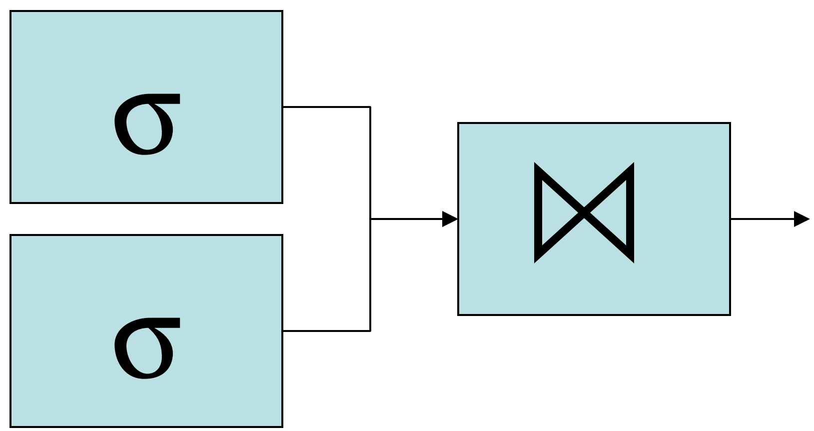
  - gossipEvent(X) :- localNode(X),
    periodic(X,E,10).

  - gossipPartner@X(X,Y) :- gossipEvent@X(X),
    neighbour@X(Y).

  - member@Y(Z) :- gossipPartner@X(X,Y),
    coinflip(weight),
    member@X(Z).

Software Dataflow Graph

- Elements represented as C++ objects
- V. efficient tuple handoff
  - Virtual fn call + refcounts
- Blocking/unblocking w/ continuations
- Single-threaded async i/o scheduler



Typical dataflow elements

- Relational operators
  - Select, join, aggregate, groupby
  - Generalised projection (PEL)
- Networking stack
  - Congestion control, routing, SAR, etc.
- "Glue" elements
  - Queues, muxers, schedulers, etc.
- Debugging
  - Loggers, watchpoints, etc.

Evaluation: Chord test case

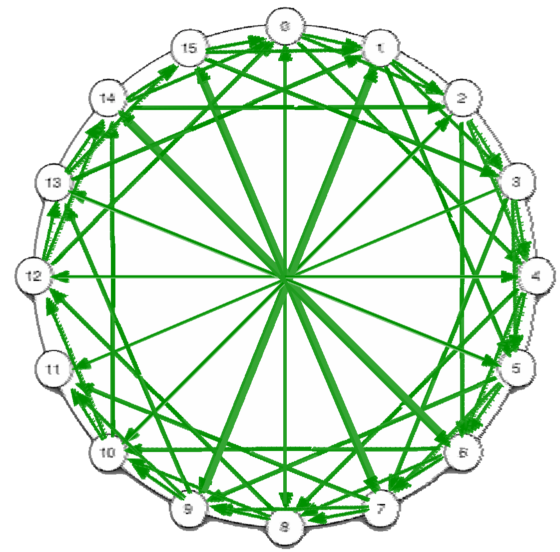
- Why Chord?
  - Quite complex overlay
  - Several different data structures
  - Maintenance dynamics, inc. churn
- Need to show:
  - We can concisely express Chord's properties
  - We can execute the specification with acceptable performance

Chord (Stoica et. al. 2001)
a "distributed hash table"

- Flat, cyclic *key space* of 160-bit identifiers
- Nodes pick a random identifier
  - E.g. SHA-1 of IP address, port
- *Owner* of key k: node with lowest ID greater than k
- Efficiently route to owner of any key in ~log(n) hops

Chord data structures

- *Predecessor* node
- *Successor set*
  - log(n) next nodes
- *Finger* table
  - Pointers to power-of-2 positions around the ring



Chord dynamics

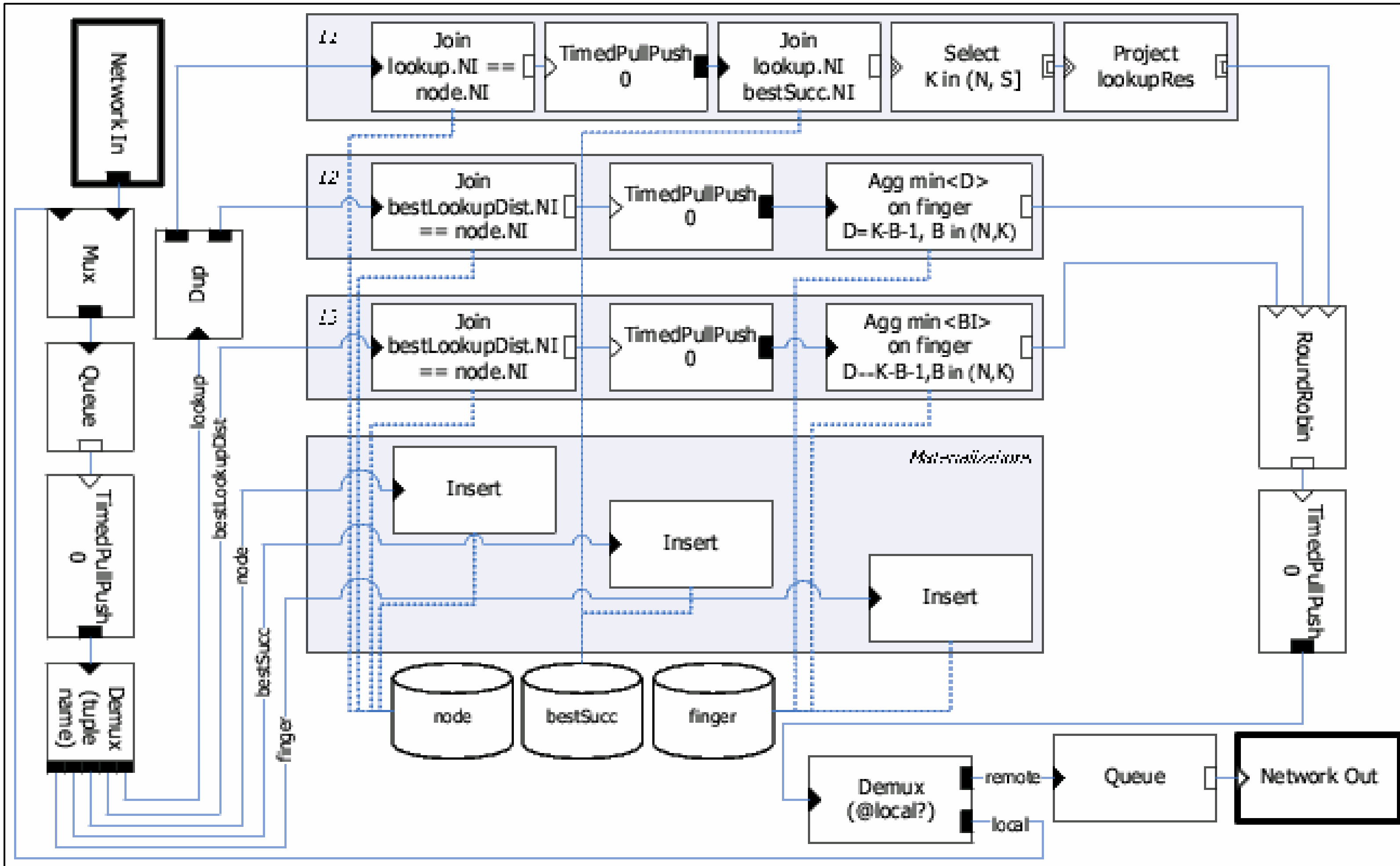
- Nodes *join* by looking up the owner of their ID
- Download successor sets from neighbours and perform lookups for fingers
- Periodically measure connectivity to successors & fingers
- *Stabilization* continously optimizes finger table

Example: Chord in 33 rules

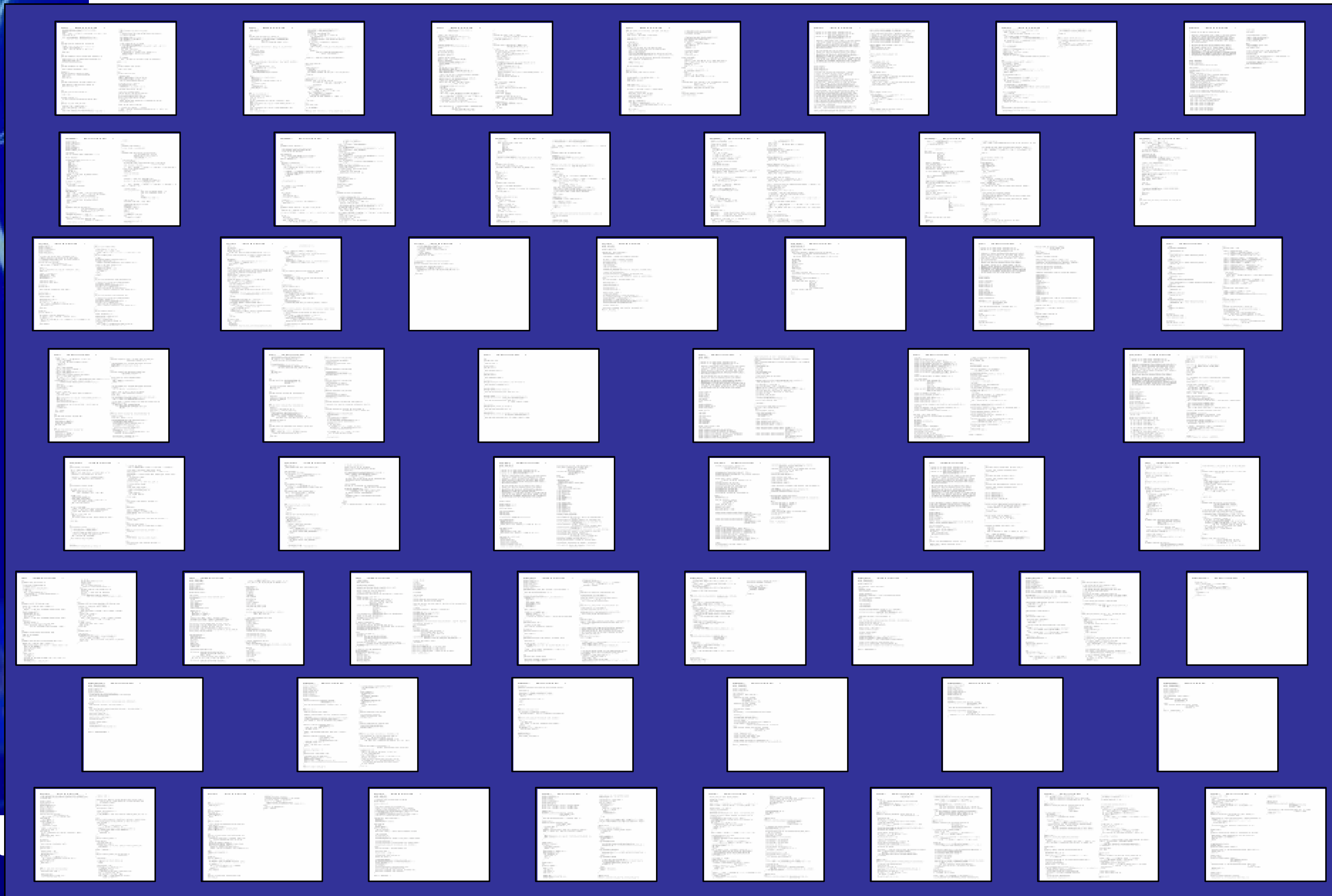
chord.plg   Sun Apr 10 23:43:49 2005   1

```
/*
 * 1.1 Chord
 * ---------
 */

/* The base tuples */
materialise(node, infinity, 1).
materialise(bestSucc, tSucc * 2, 1).
materialise(finger, tFix * 2, 160).
materialise(succ, tSucc * 2, 16).
materialise(pred, infinity, 1).
materialize(join, tJoin, 5).
materialize(landmark, infinity, 5).
materialize(stabilize, tStabilizeWait, 5).
materialize(pingNode, tPingSoftState, infinity).

/** Lookups |||3||| */
rule L1 lookupRes@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].
rule L2 bestLookupDist@NI(NI,K,R,E,min<D>) :- lookup@NI(NI,K,R,E),
        finger@NI(NI,I,B,BI), node@NI(NI,N), B in (N,K), D=K-B-1.
rule L3 lookup@BI(min<BI>,K,R,E) :- node@NI(NI, N),
        bestLookupDist@NI(NI,K,R,E,D), B in (N,K).
        finger@NI(NI,I,B,BI), D=K-B-1.

/* Neighbor Selection |||3||| */
rule SU1 bestSuccDist@NI(NI,min<D>) :- node@NI(NI,N), succ@NI(NI,S,SI),
        D=S-N-1.
rule SU2 bestSucc@NI(NI,S,SI) :- succ@NI(NI,S,SI),
        bestSuccDist@NI(NI,D), node@NI(NI,N), D=S-N-1.
rule SU3 finger@NI(NI,0,S,SI) :-
        bestSuccessor@NI(NI,S,SI).

/* Successor eviction |||4||| */
rule SR1 succCount(NI,count<>) :- succ(NI,S,SI).
rule SR2 evictSucc@NI(NI) :- succCount@NI(NI,C), C >
        succSize.
rule SR3 maxSuccDist@NI(NI,max<D>) :- succ@NI(NI,S,SI),
        node@NI(NI,N), D = f_dist(N,S)-1, evictSucc@NI(NI).
rule SR4 delete@NI<succ(NI,S,SI)> :- succ@NI(NI,S,SI),
        maxSuccDist@NI(NI,D), D=f_dist(N,S)-1.

/* Finger fixing |||3||| */
rule F1 fFix@NI(NI,E,I) :- periodic@NI(NI,E,t_Fix), I in [0, fNum),
        f_coinFlip(fFixProb).
rule F2 lookup@NI(NI,K,NI,E) :- fFix@NI(NI,E,I), node(NI,N), K = N + 1
        << I.
rule F3 finger@NI(NI,I,B,BI) :- fFix@NI(NI,E,I),
        lookupRes@NI(NI,K,B,BI,E), K in (N + 1<<I, N), node@NI(NI,N).

/* Churn Handling |||5||| */
rule J1 pred@NI(ni,null,"").
rule J2 joinReq@LI(LI,N,NI,E) :- join@NI(NI,E), node@NI(NI,N),
        landmark@NI(NI,LI), LI != "".
rule J3 succ@NI(NI,N,NI) :- landmark@NI(NI,LI), node@NI(NI,N),
        join@NI(NI,E), LI = "".
rule J4 lookup@LI(LI,N,NI,E) :- joinReq@LI(LI,N,NI,E).
rule J5 succ@NI(NI,S,SI) :- join@NI(NI,E), lookupRes@NI(NI,K,S,SI,E).

/* Stabilization |||8||| */
rule S0 stabilize@NI(NI, E) :- periodic@NI(NI, E, t_stab).
rule S1 stabilizeReq@SI(SI,NI,E) :- stabilize@NI(NI,E),
        bestSucc@NI(NI,S,SI).
rule S2 sendPred@PI1(PI1,P,PI,E) :- stabilizeReq@NI(NI,PI1,E),
        pred@NI(NI,P,PI), PI != "-".
rule S3 succ@NI(NI,P,PI) :- node(NI,N), sendPred@NI(NI,P,PI,E),
        bestSucc@NI(NI,S,SI), P in (N,S), stabilize@NI(NI,E).
rule S4 sendSucc@SI(SI,NI) :- stabilize@NI(NI, E),
        succ@NI(NI,S,SI).
rule S5 succ@PI(PI,S,SI) :- sendSucc@NI(NI,PI), succ@NI(NI,S,SI).
rule S6 notifyPred@SI(SI,N,NI) :- stabilize@NI(NI, E), node@NI(NI,N),
        successor@NI(NI,S,SI).
rule S7 pred@NI(NI,P,PI) :- node@NI(NI,N), notifyPred@NI(NI,P,PI),
        pred@NI(NI,P1,PI1), ((PI1 == "") || (P in (P1, N))).

/** Connectivity Monitoring */
rule C1 pingReq@PI(PI, NI, E, TS) :- periodic@NI(NI, E, tPing),
        pingNode@NI(NI, PI), TS = f_currentTime().
rule C2 pingResp@RI(RI, NI, E, TS) :- pingReq@NI(NI, RI, E, TS).
rule C3 latency@NI(NI, PI, L) :- pingReply@NI(NI, PI, E, TS),
        pingReq@NI(NI, PI, E, TS1), TS2 = f_currentTime(), L = TS2 -
        TS1.

rule CS1 pingNode@NI(NI,SI) :- succ@NI(NI,S,SI).
rule CS2 succ@NI(NI, S, SI) :- succ@NI(NI, S, SI), latency@NI(NI, SI,
        L).

rule CF1 pingNode@NI(NI,FI) :- finger@NI(NI,I,B,BI).
rule CF2 finger@NI(NI, I, B, BI) :- finger@NI(NI, I, B, BI),
        latency@NI(NI, BI, L).
```

Dataflow graph
(some of it, at least)



Comparison: MIT Chord in C++



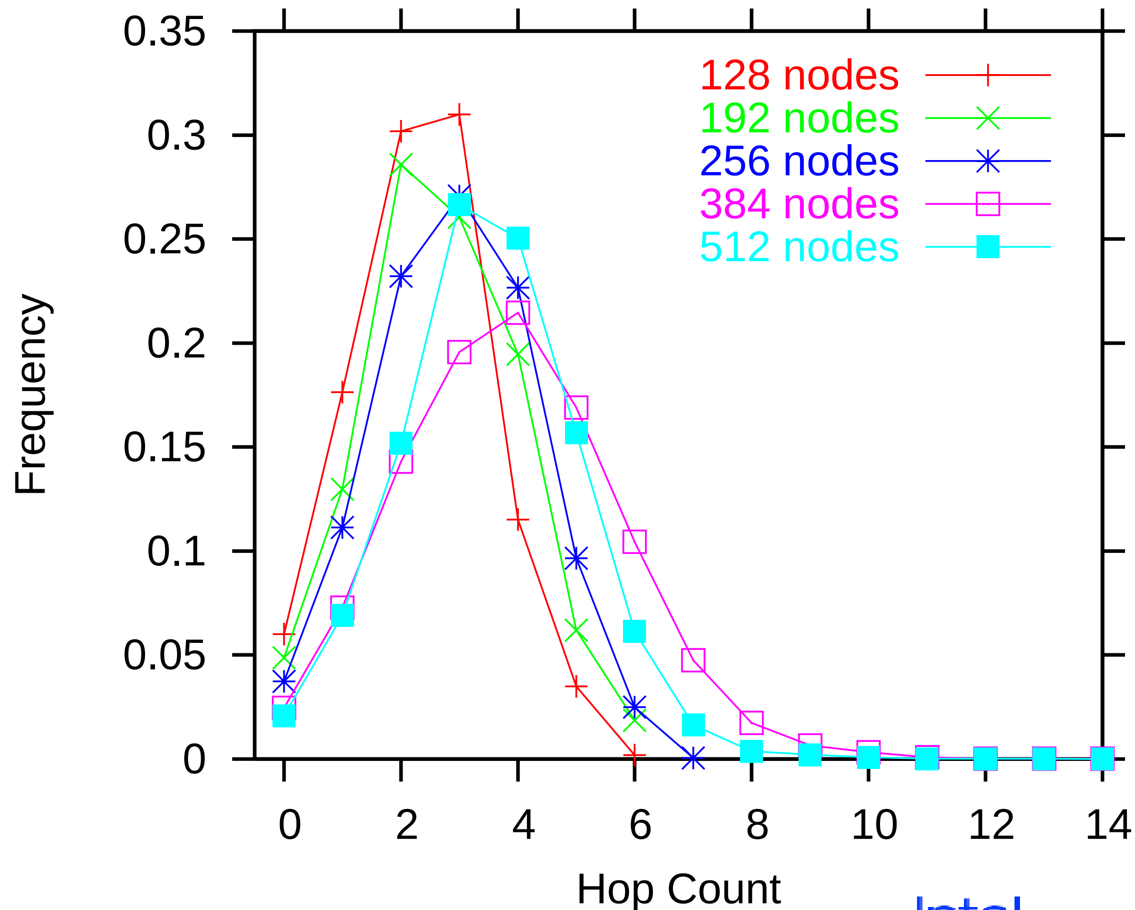
Perhaps a fairer comparison…

- Macedon (OSDI 2004)
  - State machines, timers, marshaling, embedded C++
- Macedon Chord: 360 lines
  - 32-bit IDs, no stabilization, single successor
- P2 Chord: 34 lines
  - 160-bit IDs, full stabilization, log(n) successor sets, optimized

Performance?

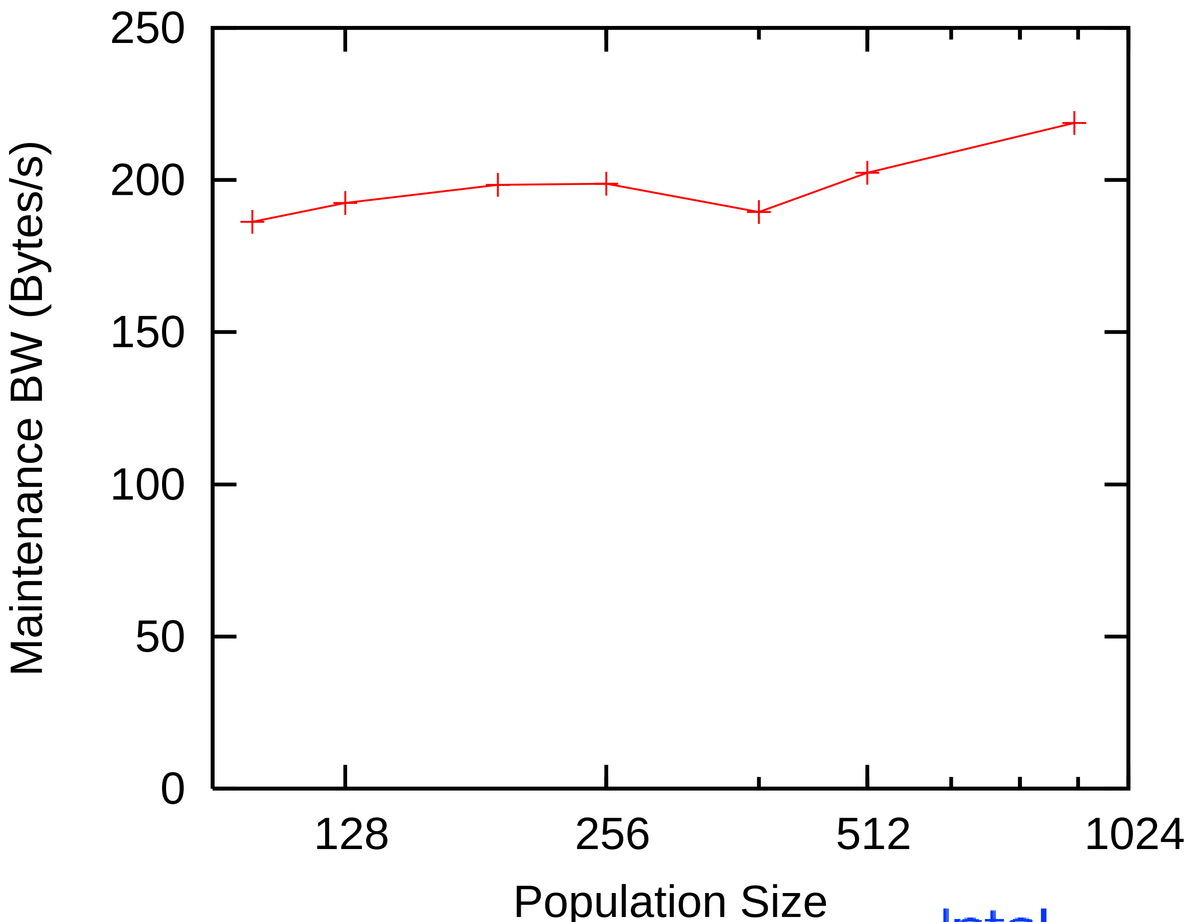
- Note: aim is *acceptable* performance, not necessarily that of hand-coded Chord
- Analogy: SQL / RDBM systems

Lookup length in hops



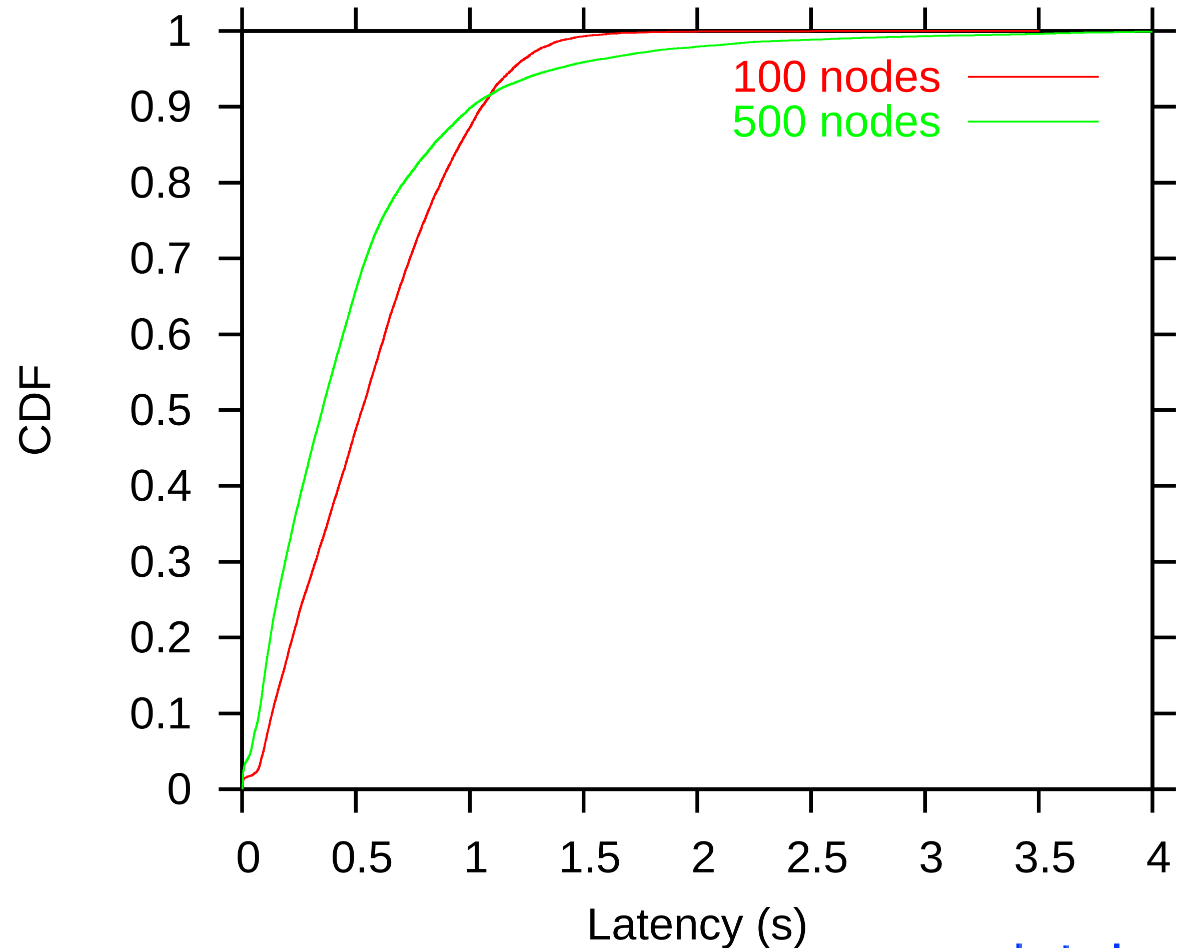
Intel Research Berkeley

Maintenance bandwidth
(comparable with MIT Chord)



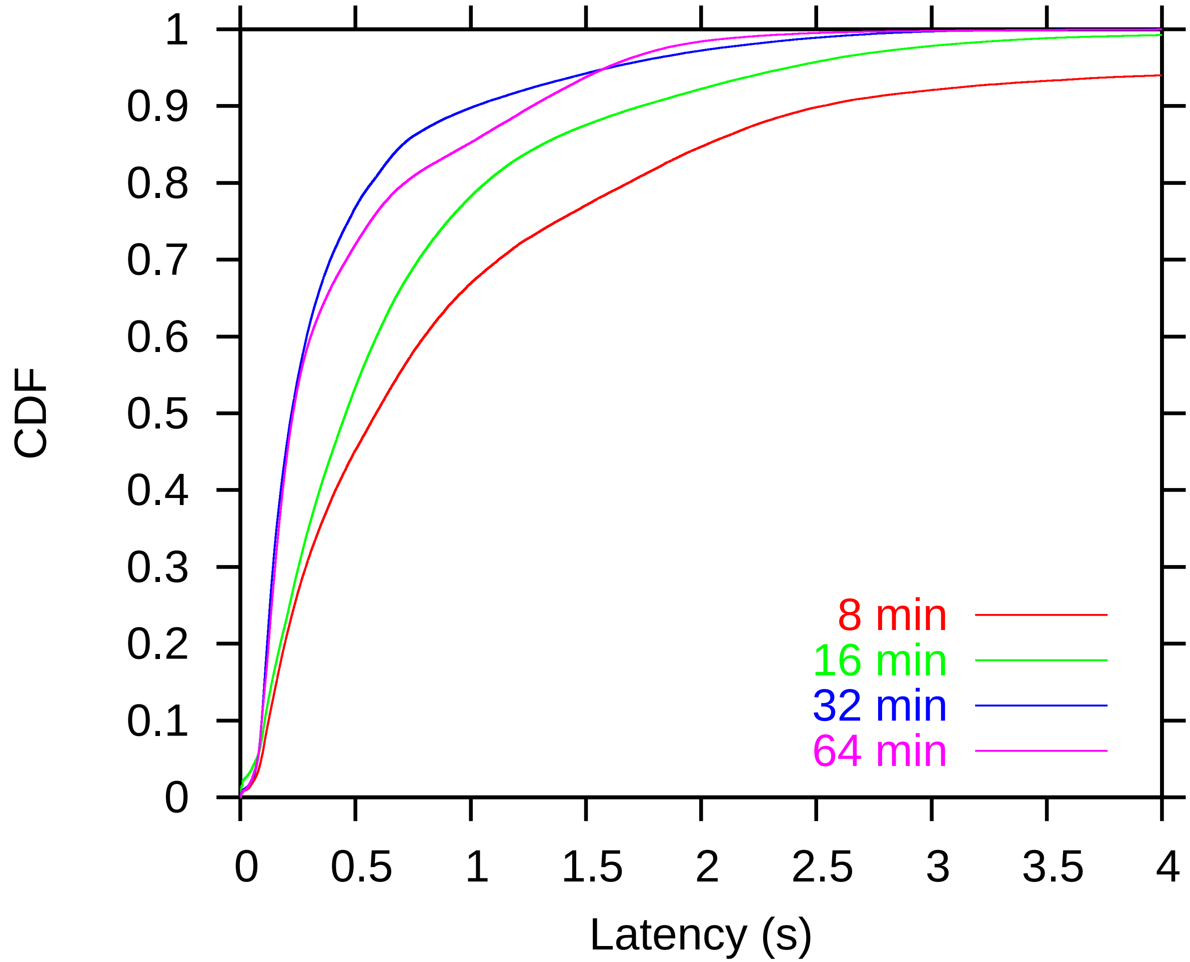
Intel Research Berkeley

Latency without churn



Intel Research Berkeley

Latency under churn



Intel Research Berkeley

Ongoing work

- More overlays!
  - Pastry, parameterized small-world graphs
  - Link-state, distance vector algorithms
  - Assorted multicast graphs
- Proper library interface
  - Code release later this summer
- Integrate discovery
  - Exploit power of full query processor
  - Can implement PIER in P2
  - Integrated management, monitoring, measurement

Ongoing work

- Rich seam for further research!
  - The "right" language (SIGMOD possibly)
  - Optimization techniques
  - Proving safety properties
- Reconfigurable transport protocols
  - Dataflow framework facilitates composition
  - P2P networks introduce new space for transport protocols
- Debugging support
  - Use query processor for online distributed debugging
  - Potentially very powerful

Conclusion

- Diverse overlay networks can be expressed concisely in a OverLog
- Specifications can be directly executed by P2 to maintain the overlay
- Performance of P2 overlays remains comparable with hand-coded protocols

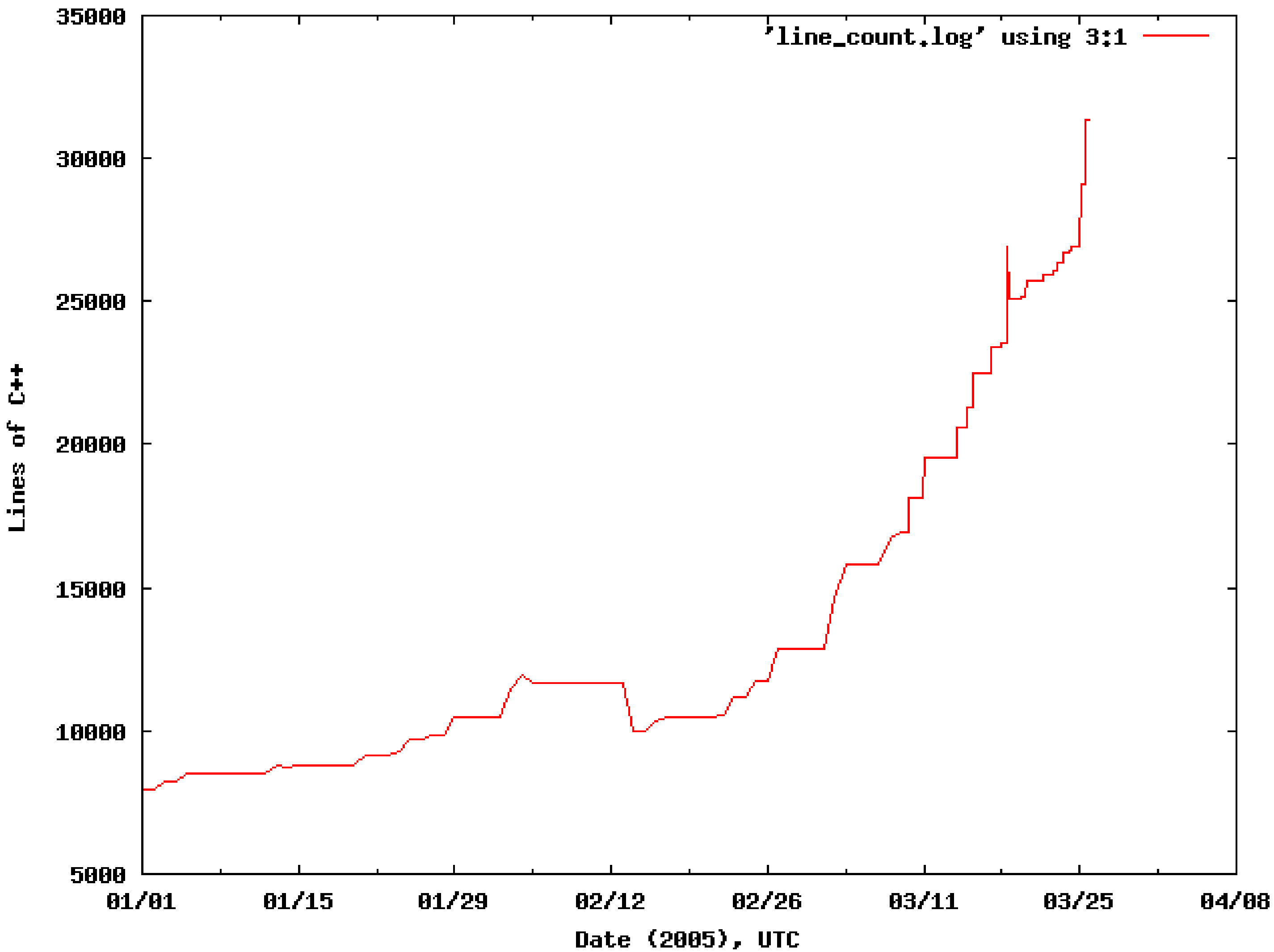
Long-term implications

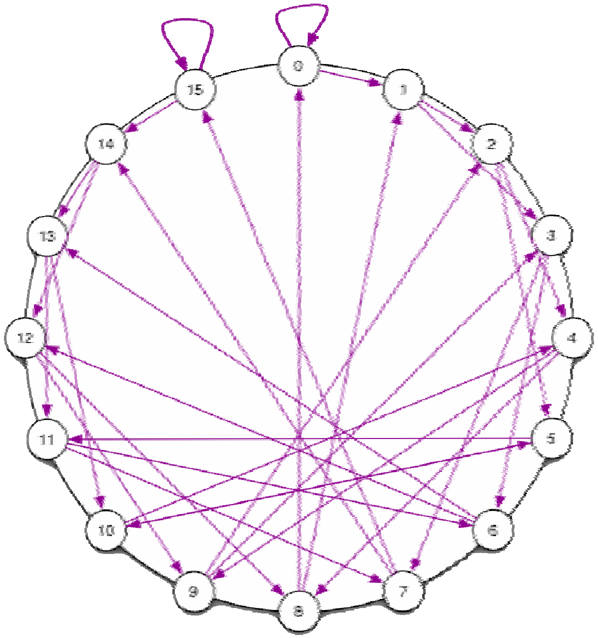
- An abstraction and infrastructure for radically rethinking networking
  - One possibility: System R for networks
- Where does the network end and the application begin?
  - E.g. can run queries to monitor the network at the endpoints
  - Integrate resource discovery, management, routing
  - Chance to reshuffle the networking deck

Thanks! Questions?

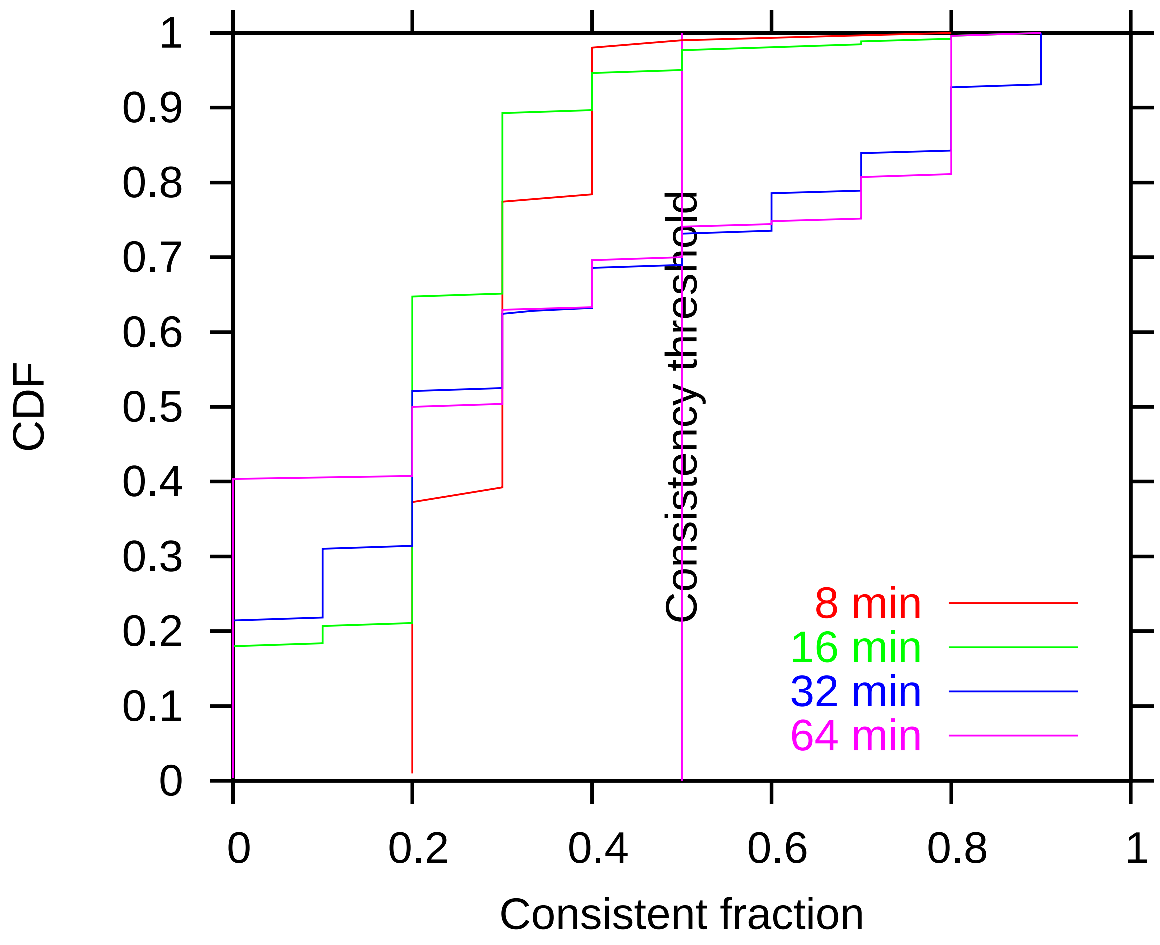
Timothy Roscoe
troscoe@acm.org

It's real…

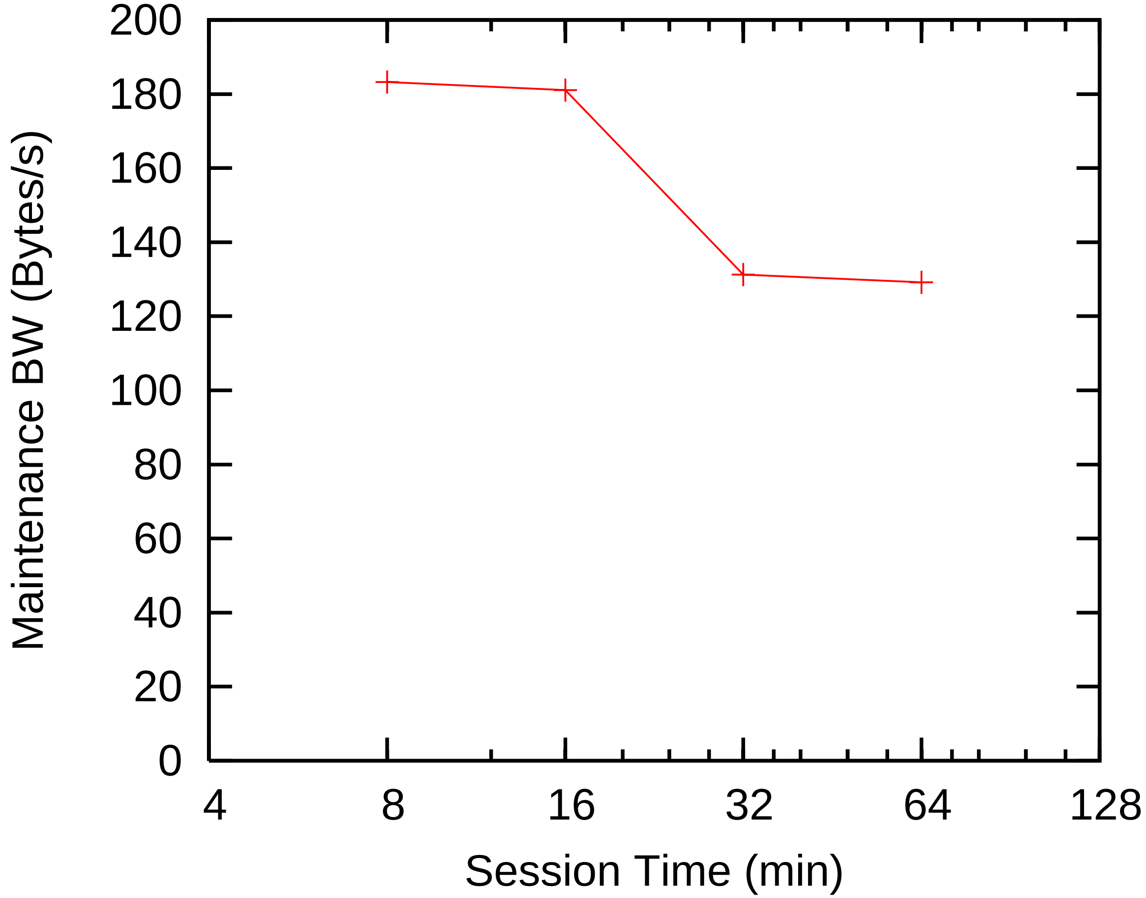

'line_count.log' using 3:1
Lines of C++
35000
30000
25000
20000
15000
10000
5000
01/01
01/15
01/29
02/12
02/26
03/11
03/25
04/08
Date (2005), UTC


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Consistency under churn



Bandwidth usage under churn



P2: A declarative overlay engine

- Everything is a declarative query
  - Overlay construction, maintenance, routing, monitoring
- Queries compiled to software dataflow graph and directly executed
- System written from scratch (C++)
  - Deployable (PlanetLab, Emulab)
  - Reasonable performance so far for deployed overlays