Declarative Routing: Extensible Routing with Declarative Queries

Boon Thau Loo* Joseph M. Hellerstein*† Ion Stoica* Raghu Ramakrishnan$^{\delta}$
*UC Berkeley †Intel Research $^{\delta}$University of Wisconsin-Madison
{boonloo, jmh, istoica}@cs.berkeley.edu raghu@cs.wisc.edu

ABSTRACT

The Internet's core routing infrastructure, while arguably robust and efficient, has proven to be difficult to evolve to accommodate the needs of new applications. Prior research on this problem has included new hard-coded routing protocols on the one hand, and fully extensible Active Networks on the other. In this paper, we explore a new point in this design space that aims to strike a better balance between the extensibility and robustness of a routing infrastructure. The basic idea of our solution, which we call *declarative routing*, is to express routing protocols using a database query language. We show that our query language is a natural fit for routing, and can express a variety of well-known routing protocols in a compact and clean fashion. We discuss the security of our proposal in terms of its computational expressive power and language design. Via simulation, and deployment on PlanetLab, we demonstrate that our system imposes no fundamental limits relative to traditional protocols, is amenable to query optimizations, and can sustain long-lived routes under network churn and congestion.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design

General Terms

Design, Languages, Security

Keywords

Extensible routing, Declarative queries, Routing languages

1. INTRODUCTION

Designing routing protocols is a difficult process. This is not only because of the distributed nature and scale of the networks, but also because of the need to balance the extensibility and flexibility of these protocols on one hand, and their robustness and efficiency on the other hand. One need look no further than the Internet for an illustration of these hard tradeoffs. Today's Internet routing protocols, while arguably robust and efficient, are hard to change to accommodate the needs of new applications such as improved resilience and higher throughput. Upgrading even a single router is hard [16]. Getting a distributed routing protocol implemented correctly is even harder. And in order to change or upgrade a deployed routing protocol today, one must get access to *each* router to modify its software. This process is made even more tedious and error prone by the use of conventional programming languages.

Several solutions have been proposed to address the lack of flexibility and extensibility in Internet routing. Overlay networks allow third parties to replace Internet routing with new, "from-scratch" implementations of routing functionality that run at the application layer. However, overlay networks simply move the problem from the network to the application layer where third parties have control: implementing or updating an overlay routing protocol still requires a complete protocol design and implementation, and requires access to the overlay nodes. A radically different approach, Active Networks [14], allows one to deploy new routing functionality without the need to have direct access to routers. However, due to the general programming models proposed for Active Networks, they present difficulties in both router performance and the security and reliability of the resulting infrastructure.

In this paper, we explore a new point in this design space that aims to strike a better balance between the extensibility and the robustness of a routing infrastructure. Our solution, which we call *declarative routing*, can be viewed as an application of database techniques to the domain of networking. It is based on the observation that recursive query languages studied in the deductive database literature [5, 23] are a *natural* fit for expressing routing protocols. Deductive database query languages focus on identifying recursive relationships among nodes of a graph, and are well suited for expressing paths among nodes in a network.

With declarative routing, a routing protocol is implemented by writing a simple query in a declarative query language like *Datalog*, which is then executed in a distributed fashion at some or all of the nodes. Declarative routing can be viewed as a restrictive instantiation of Active Networks for the control plane, which aims to balance the concerns of expressiveness, performance and security, properties which are needed for an extensible routing infrastructure to succeed. Next, we discuss how declarative routing satisfies these properties:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCOMM'05, August 21–26, 2005, Philadelphia, Pennsylvania, USA.
Copyright 2005 ACM 1-59593-009-4/05/0008 ...$5.00.

Expressiveness: As we will show in Section 3, Datalog queries can express a variety of well-known routing protocols (*e.g.*, distance vector, path vector, dynamic source routing, link state, multicast) in a compact and clean fashion, typically in a handful of lines of program code. Moreover, higher-level routing concepts (*e.g.*, QoS constraints) can be achieved via simple modifications to these queries. Finally, writing the queries in Datalog illustrates surprising relationships between protocols. In particular, we show that distance vector and dynamic source routing differ only in a simple, traditional query optimization decision: the order in which a query's predicates are evaluated.

Efficiency: By leveraging well-studied query optimization techniques (in Section 7), we show (in Section 9) via simulation and implementation that there is no inherent overhead in expressing standard protocols via a declarative query language. In addition, query optimization techniques can lead to efficient execution, and facilitate work-sharing among queries.

Security: As shown in Section 6, Datalog has several desirable security properties. In particular, Datalog is a side-effect free language, and Datalog queries can be easily "sandboxed". Furthermore, the complexity of basic Datalog queries is polynomial in the size of the input [5]. While adding functions to Datalog alters its theoretical worst-case complexity, powerful tests for termination on given inputs are available [18].

Declarative routing could evolve to be used in a variety of ways. One extreme view of the future of routing is that individual end-users (or their applications) will explicitly request routes with particular properties, by submitting route construction queries to the network. The safety and simplicity of declarative queries would clearly be beneficial in that context. A more incremental view is that an administrator at an ISP might reconfigure the ISP's routers by issuing a query to the network; different queries would allow the administrator to easily implement various routing policies between different nodes or different traffic classes. Even in this managed scenario, the simplicity and safety of declarative routing has benefits over the current relatively fragile approaches to upgrading routers. While this second scenario is arguably the more realistic one, in this paper, we consider the other extreme in which any node (including end-hosts) can issue a query. We take this extreme position in order to explore the limits of our design.

To demonstrate the feasibility of our idea, we have implemented a prototype on top of PIER [2], a distributed relational query processor. Through a combination of simulations on transit-stub network topologies and experiments on the PlanetLab [21] testbed, we evaluate the scalability and efficiency of our system, as well as its ability to sustain long-lived routes under network churn and congestion.

We do not propose that this work in its current form can serve as a "drop-in" replacement for existing network infrastructures, which have evolved and coagulated over many years under various constraints. However, if progress is to be made in deploying new, more flexible infrastructures for network routing, we believe that a cleaner foundation is needed. Our work can be viewed as a step in that direction, based on applying well-studied techniques from the deductive database literature to the network domain.

The remainder of this paper is organized as follows. First, we provide an overview of our system model in Section 2. In Section 3, we introduce the basic concepts of query specification and execution, by stepping through a concrete example using the familiar path vector and distance vector protocols. Next, we will illustrate the flexibility of our query language through several examples in Section 5. We then address the challenges of security (Section 6), query optimization techniques for efficiency (Section 7) and route stability and robustness (Section 8). Following our evaluation (Section 9) in simulations and on PlanetLab, we conclude with related (Section 10) and future work (Section 11).

2. SYSTEM MODEL

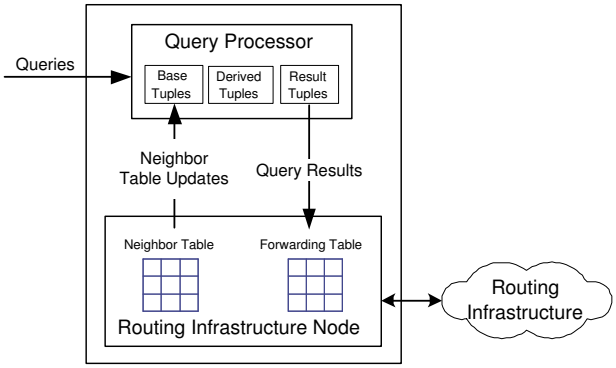

Figure 1: ***Basic Components of Routing Infrastructure Node and Co-located Query Processor.***

We model the routing infrastructure as a directed graph, where each link is associated with a set of parameters (*e.g.*, loss rate, available bandwidth, delay). The nodes in the routing infrastructure can either be IP routers or overlay nodes.

In a *centralized* design such as the Routing Control Platform [15], network information is periodically gathered from the routing infrastructure, and stored at one or more central servers. Each query is sent to one or more of these servers, which process the queries using their internal databases and set up the forwarding state at the routers in the network.

In this paper, we focus on a *fully distributed* implementation to explore the limits of our design. Like traditional routers, the infrastructure nodes in our model maintain links to their neighbors, compute routes, and set up the forwarding state to forward data packets. However, instead of running a traditional routing protocol, each infrastructure node runs a general-purpose query processor.

Figure 1 shows the basic components of an infrastructure node (*router*) and its co-located query processor. Each router maintains a typical set of local information including the links to its neighbors (*neighbor table*) and the forwarding information to route packets (*forwarding table*). The neighbor table is periodically updated in response to link failures, new links, or link metric changes. These updates are performed outside the query processor using standard mechanisms such as periodic pings. The query processor can read the neighbor table (either periodically or upon being notified of updates), and install entries into the forwarding table. In our discussion, this simple interface is the only interaction required between the query processor and the router's core forwarding logic.

Both routing protocols and route requests can be expressed as declarative queries, and issued either by the routers themselves or by third-parties or end-hosts. Upon receiving the query request, each query processor initiates a distributed ex-

ecution of the query in the network. The results of the query are used to establish router forwarding state (the *forwarding table* in Figure 1), which the routers use for forwarding data packets. Alternatively, the query results can be sent back to the party that issued the query, which can use these results to perform source routing.

During query execution, intermediate data generated by a query processor is stored locally, and can be sent to neighboring query processors for further processing. To keep with the database terminology, we refer to the local information that the node reads as *base tuples*, and the generated intermediate data as *derived tuples*. Tuples are organized in *tables*. The base tuple we use frequently in our examples is the *link* tuple: **link(<u>source</u>, destination, ...)**. A *link* tuple corresponds to a copy of an entry in the neighbor table, and represents an edge from the node itself (*source*) to one of its neighbors (*destination*). The other fields, "...", represent link metrics (delay, loss rate and bandwidth). Each tuple is stored at the address indicated by the <u>underlined</u> address field. Thus, in the case of a link tuple, <u>*source*</u> represents the address of the node storing the neighbor table. Each link tuple is uniquely identified by its source and destination pair (the *unique key*).

An example of a derived tuple is the *path* tuple: **path(<u>source</u>, destination, pathVector, cost)**. A *path* tuple indicates that *destination* is reachable from *source* along the path in *pathVector*, where *cost* is the aggregate sum of all link costs along the path. Each path tuple is uniquely identified by its pathVector.

In addition, some of the derived tuples generated by the query processor also form part of the query answer as *result tuples* as specified by the query. In Sections 3 and 5 we will see examples of result tuples *path*, *nextHop* and *bestPath*.

Each query is accompanied by a specification of the lifetime (duration) of the protocol/route. During the lifetime of a query, neighbor table updates are made available to the query processor, and these updates trigger the recomputation of some of the existing derived and result tuples. In the next section, we step through a concrete example of how a query can be specified and executed within this execution model.

3. THE BASICS

We will start with a query that expresses the transitive closure of the *link* table, which can be used to compute network reachability. We will show how this query can be expressed in Datalog, describe the generation of a query plan for this query, and finally show the execution of the query plan on a distributed query engine. We illustrate the connection between recursive queries and routing protocols by showing that the execution of this query resembles the well-known path vector and distance vector routing protocols.

3.1 Datalog Program Syntax

Datalog is similar to Prolog [13], but hews closer to the spirit of declarative queries, exposing no imperative control. A Datalog *program* consist of a set of declarative *rules*. Since these programs are commonly called *"recursive queries"* in the database literature, we will use the term "query" when we refer to a Datalog program. A Datalog *rule* has the form <*head*> :- <*body*>, where the *body* is a list of predicates over constants and variables, and the *head* defines a set of tuples derived by variable assignments satisfying the body's predicates. The rules can refer to each other in a cyclic fashion to express recursion. The order in which the rules are presented is immaterial. The commas separating the predicates in a rule are logical conjuncts (*AND*), and the order in which predicates appear has no semantic significance.

Following Prolog-like conventions [23], names for tuples, predicates, function symbols and constants begin with a lower-case letter, while variable names begin with an upper-case letter. Most implementations of Datalog enhance it with a limited set of function calls (which start with *"f_"* in our syntax), including boolean predicates, arithmetic computations and simple list manipulation (*e.g.*, the $f_concatPath$ function in our first example). Presented with a program, a Datalog system will find all possible assignments of tuples to unbound variables that satisfy the rules in the query.

3.2 First Datalog Example

Our first example, the *Network-Reachability* query, takes as input link tuples, and computes the set of all paths encoded in path tuples. In all our examples, S, D, C and P abbreviate the *source*, *destination*, *cost* and *pathVector* fields respectively for both the link and path tuples. As before, the address fields indicating the network storage location of the tuples are <u>underlined</u>. We begin our discussion by looking only at the part of the query written in *bold* text, ignoring the rest of the text for a moment.

NR1: **path(<u>S</u>,D**,P,C) :- **link(<u>S</u>,D**,C),
P = $f_concatPath$(link(<u>S</u>,D,C), nil).

NR2: **path(<u>S</u>,D**,P,C) :- **link(<u>S</u>,Z**,C_1), **path(<u>Z</u>,D**,P_2,C_2),
C = C_1 + C_2,
P = $f_concatPath$(link(<u>S</u>,Z,C_1),P_2).

Query: **path(<u>S</u>,D**,P,C).

Rule NR1 produces one-hop paths from existing link tuples, storing them at the source node. Rule NR2 recursively produces path tuples of increasing cost by matching the destination fields of existing links to the source fields of previously computed paths. The rule *Query* specifies the output of interest (i.e. result tuples), which are the path tuples stored at the source node. The matching is expressed using the two "Z" fields in $link(\underline{S}, Z, C_1)$ and $path(\underline{Z}, D, P_2, C_2)$ in rule NR2. Intuitively, rule NR2 says that if there is a link from node S to node Z, and there is a path from node Z to node D, then there is a path from node S to node D via Z.

The query does not impose any restriction on either source or destination as both S and D are unbound variables. Hence, the query computes the *full transitive closure* consisting of the paths between all pairs of reachable nodes. If the query is only interested in the paths from a given node b to every other node in the network, the query would be $path(\underline{b}, D, P, C)$, with the source field bound to constant b.

We now focus on the remaining portions of rules NR1 and NR2. The expression $P = f_concatPath(L, P_1)$ is a predicate function that is satisfied if P is assigned to the path vector produced from prepending link L to the existing path vector P_1. With these additions, rules NR1 and NR2 also compute the total path costs, and the path vectors.

The above query will not terminate due to the generation of path tuples with cycles. To prevent computing paths with cycles, we can add an extra predicate $f_inPath(P_2, S) = false$ to rule NR2, where the function $f_inPath(P, S)$ returns true if node S is in the path vector P.

3.3 Query Plan Generation

To execute a query, we first need to generate a query plan.

A query plan is a dataflow diagram consisting of relational operators that are connected by arrows indicating the flow of tuples during the query execution. Figure 2 shows a query plan for the Datalog query discussed in the previous section. The query plan is formulated based on Datalog's *semi-naïve* fixpoint evaluation [8] mechanism which ensures that each rule does not redundantly generate the same tuple twice.

Rule NR1 is implemented by simply renaming existing link tuples to path tuples. This is shown by the rightward arrow at the bottom of the figure from $link(\underline{S}, D, C)$ to $path(\underline{S}, D, P, C)$.

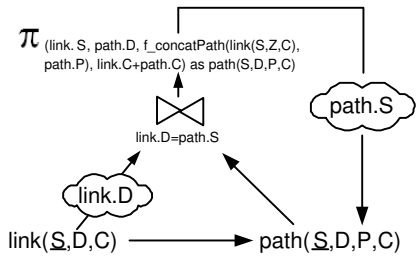

Figure 2: ***Query Plan for the Network-Reachability Query.***

Rule NR2 requires a relational join ⋈) operator to match the destination fields of link tuples ($link.D$) with the source fields of existing path tuples ($path.S$). For each pair of link and path tuples that matches, the join operator produces a new path tuple that is the concatenation of the original path and link tuples. The projection π operator takes as input the output of the join and a list of fields, extracts these fields from the join's output, and optionally renames them. This ensures that only the required path fields specified in the query are generated as path tuples. Unlike most textbook query plans, this dataflow forms a cycle, which captures the recursive use of the *path* rule definition in the query.

The clouds represent the forwarding of tuples from one network node to another, and are labeled with the destination node. The first cloud ($link.D$) ships link tuples to the neighbor nodes indicated by their destination address fields, in order to join with matching path tuples stored by their source address fields. The second cloud ($path.S$) ships new path tuples computed from the join back to their neighboring source nodes for further processing.

3.4 Query Plan Execution

Next, we focus on the distributed execution of a routing query. To simplify the exposition, we will temporarily ignore the mechanism for initially disseminating the Datalog query to the network nodes; we return to this issue in Section 3.5.

Upon receipt of the Datalog query, each node creates the query plan shown in Figure 2 and starts executing the plan for the duration of the query. When the query plan is executed, the flow of tuples in the network enables nodes to exchange the routing information necessary to compute the queried paths. Figure 3 shows the tuples that are generated during the execution of the query plan in Figure 2 for a simple network consisting of five nodes. $p(\underline{S}, D, P, C)$ abbreviates $path(\underline{S}, D, P, C)$. Link costs in our example are set to 1, and hence path cost is equal to the number of hops. $l'(S, \underline{D}, C)$ refers to link tuples that are sent by node S and cached at destination node D. We show only the new path tuples (in *bold*) generated at each iteration.

For clarity, we describe the communication in stages, where

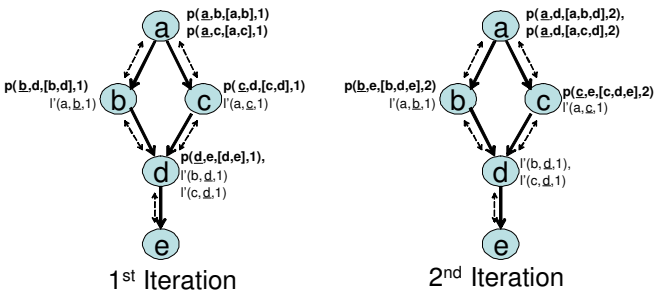

Figure 3: ***Nodes in the network are running the query plan in Figure 2. The dashed lines represent the control plane (along which tuples are sent), while the full lines represent the data plane (along which data packets are forwarded).***

each stage or *iteration* represents a "round of communication", in which all nodes exchange tuples from the previous iteration. The rounds of communication is a simplification of actual query execution: since dataflow is fully asynchronous, tuples for the next round can be generated as soon as tuples from the previous round are computed. Each iteration represents the traversal of a "cloud" in Figure 2. The first iteration derives single-hop path tuples from the first rule of the query. It does this by traversing the $link.D$ cloud, which ships link tuples to the address in their destination field, where they are cached for the duration of the query (denoted by $l'(S, \underline{D}, C)$ in Figure 3)[1]. Since the query has no recursion on the *link* table, all subsequent iterations involve the other cloud ($path.S$).

In the 2nd iteration, the link tuples are joined with existing one-hop path tuples to produce two-hop path tuples. These tuples are then sent back to the source nodes (the $path.S$ cloud) and three-hop path tuples are computed. Once the query reaches a node, the node takes up to k iterations to converge to a steady state, where k is the diameter of the network. Including the initial query dissemination which takes up to k iterations to reach the node furthest from the query node, the total time taken for the query to converge is proportional to $2k$. To illustrate further, we step through the communication necessary for the computing the path tuple $p(\underline{a}, d, [a, c, d], 2)$ for node a:

1st **iteration:** Node a ships $l(\underline{a}, c, 1)$ to c (via the $link.D$ cloud running at node a). It is stored as $l'(a, \underline{c}, 1)$ at node c for the duration of the query.

2nd **iteration:** Node c receives $l'(a, \underline{c}, 1)$ and performs the join of $l'(a, \underline{c}, 1)$ and $p(\underline{c}, d, [c, d], 1)$ to produce the new path tuple
$p(a, d, [a, c, d], 2)$. This new tuple is sent back to node a (the $path.S$ cloud running at node c).

3.5 Query Dissemination

Queries can be disseminated to nodes in a variety of ways. In static scenarios, the query may be "baked in" to another artifact – *e.g.*, router firmware or peer-to-peer application software. More flexibly, the query could be disseminated upon initial declaration. It may be sufficient to perform dissemination via flooding, particularly if the query will be long-lived, amortizing the cost of the initial flood. As an optimization,

[1] As an optional optimization for undirected graphs, the operation of shipping link tuples can be avoided by adding an extra rule link(S,D,C) :- link(D,S,C).

instead of flooding the query in the network, we can instead "piggy-back" dissemination onto query execution: the query can be embedded into the first data tuple sent to each neighboring node as part of the query computation. The piggy-back mechanism has the advantage that nodes that are not involved in the query computation will not receive the query. These scenarios arise in some of the examples in Section 5 below, such as creating paths that avoid certain nodes.

3.6 Path Vector or Distance Vector Protocol

The computation of the above query resembles the computation of the routing table in a path vector or distance vector protocol. The computation starts with the source computing its initial reachable set (which consists of all neighbors of the source) and shipping it to all its neighbors. In turn, each neighbor updates the reachable set with its own neighborhood set, and then forwards the resulting reachable set to its own neighbors. With minor modifications to our previous query (modifications in *bold*), the following *Distance-Vector* query expresses the distance vector computation:

DV1: path(S,D,**D**,C) :- link(S,D,C).
DV2: path(S,D,**Z**,C) :- link(S,Z,C_1),
path(Z,D,**W**,C_2), C = C_1 + C_2.
DV3: **shortestCost(S,D,*min*<C>) :- path(S,D,Z,C).**
DV4: **nextHop(S,D,Z,C) :- path(S,D,Z,C),**
shortestCost(S,D,C).
Query: nextHop(S,D,Z,C).

Aggregate constructs are represented as functions with arguments within angle brackets (<>). DV1 and DV2 are modified from the original rules NR1 and NR2 to ensure that the path tuple maintains only the next hop on the path, rather than the entire path vector itself[2]. DV3 and DV4 are added to set up routing state in the network: *nextHop(S,D,Z,C)* is stored at node *S*, where *Z* is the next hop on the shortest path to node *D*. The main difference between this query and the actual distance vector computation is that rather than sending individual path tuples between neighbors, the traditional distance vector method batches together a vector of costs for all neighbors.

By making a modification to rule DV2 and adding rule DV5, we can apply the well-known *split-horizon with poison reverse* [20] fix to the count-to-infinity problem:

#include(DV1,DV3,DV4)
DV2: path(S,D,Z,C) :- link(S,Z,C_1),
path(Z,D,W,C_2), C = C_1 + C_2, **W ≠ S.**
DV5: **path(S,D,Z,∞) :- link(S,Z,C_1), path(Z,D,S,C_2).**
Query: nextHop(S,D,Z,C).

#include is a macro used to include earlier rules. Rule DV2 expresses that if node *Z* learns about the path to *D* from node *S*, then node *Z* does not report this path back to to *S*. Rule DV5 expresses that if node *Z* receives a path tuple with destination *D* from node *S*, then node *Z* will send a path with destination *D* and infinite cost to node *S*. This ensures that node *S* will not eventually use *Z* to get to *D*.

4. CHALLENGES

We have identified four challenges that need to be addressed in justifying the feasibility of declarative routing:

Expressiveness: How expressive and flexible is the Datalog language in expressing various routing policies? What are the limitations of this language?
Security: Is Datalog safe enough to execute queries issued by untrusted third-parties?
Efficiency: Can Datalog queries be executed efficiently in a distributed system? The answer to this question hinges on two sub-questions. The first is about raw performance: can plan generation techniques be adapted or developed to enable Datalog queries to perform well in a large network system? The second is about the feasibility of exploiting our extensible framework: given that we allow many routing queries to be issued concurrently, can we significantly reduce the redundant work performed by these concurrent queries?
Stability and Robustness: Given that the network is dynamic, how can we efficiently maintain the robustness and accuracy of long term routes?

We address these challenges in the next four sections.

5. EXPRESSIVENESS

To highlight the expressiveness of Datalog, we provide several examples of useful routing protocols expressed as queries. Our examples range from well-known routing protocols (distance vector, dynamic source routing, multicast, etc.) to higher-level routing concepts such as QoS constraints. This is by no means intended to be an exhaustive coverage of the possibilities of our proposal. Our main goal here is to illustrate the natural connection between recursive queries and network routing, and to highlight the flexibility, ease of programming, and ease of reuse afforded by a query language. We demonstrate that routing protocols can be expressed in a few Datalog rules, and additional protocols can be created by simple modifications (in **bold**) to previous examples.

5.1 Best-Path Routing

We start from the base rules NR1 and NR2 used in our first *Network-Reachability* example from Section 3. That example computes *all-pairs paths*. In practice, a more common query would compute *all-pairs shortest paths*. By modifying NR2 and adding rules BPR1 and BPR2, the following *Best-Path* query generalizes the all-pairs shortest paths computation, and computes the best paths for any path metric *C*:

#include(NR1)
NR2: path(S,D,P,C) :- link(S,Z,C_1),
path(Z,D,P_2,C_2), **C = f_compute(C_1,C_2)**,
P = *f_concatPath*(link(S,Z,C_1),P_2),
BPR1: **bestPathCost(S,D,AGG<C>) :- path(S,D,P,C).**
BPR2: **bestPath(S,D,P,C) :- bestPathCost(S,D,C),**
path(S,D,P,C).
Query: bestPath(S,D,P,C).

We have left the aggregation function (*AGG*) unspecified. By changing *AGG* and the function *f_compute* used for computing the path cost *C*, the *Best-Path* query can generate best paths based on any metric including link latency, available bandwidth and node load. For example, if the query is used for computing the shortest paths, *f_sum* is the appropriate replacement for *f_compute* in rule BPR1, and *min* is the replacement for *AGG*. The resulting bestPath tuples are stored at the source nodes, and are used by end-hosts to perform source routing.

The two added rules BPR1 and BPR2 do not result in extra messages being sent beyond those generated by rules NR1 and

[2]The *W* field in DV2 represents the next-hop to node *D* from intermediate node *Z*, and can be ignored by node *S* in computing its next hop to node *D*.

NR2. This is because path tuples computed by rules NR1 and NR2 are stored at the source nodes, and bestPathCost and bestPath tuples are generated locally at those nodes. Instead of computing the best path between any two nodes, this query can be easily modified to compute *all* paths, *any* path or the *Best-k* paths between any two nodes.

Similar to the *Network-Reachability* example, we can add an extra predicate $f_inPath(P_2, S) = false$ to rule NR2 to avoiding computing best paths with cycles. We can further extend the rules from the *Best-Path* query by including constraints that enforce a QoS requirement specified by end-hosts. For example, we can restrict the set of paths to those with costs below a loss or latency threshold k by adding an extra constraint $C<k$ to the rules NR1 and NR2.

5.2 Policy-Based Routing

Our previous example illustrates a typical network-wide routing policy. In some cases we may want to restrict the scope of routing, *e.g.*, by precluding paths that involve "undesirable" nodes. An example would be finding a path among nodes in an overlay network such as PlanetLab that avoids nodes belonging to untruthful or flaky ISPs. Such policy constraints can be simply expressed by adding an additional rule:

#include(NR1, NR2)
PBR1: permitPath(S,D,P,C) :- path(S,D,P,C),
excludeNode(S,W), $f_inPath(P, W) = false$.
Query: permitPath(S,D,P,C).

In this query, we introduce an additional table *excludeNode*, where $excludeNode(\underline{S}, W)$ is a tuple that represents the fact that node S does not carry any traffic for node W. This table is stored at each node S.

If BPR1 and BPR2 are included as rules, we can generate bestPath tuples that meet the above policy. Other policy based decisions include ignoring the paths reported by selected nodes or insisting that some paths have to pass through (or avoid) one or multiple pre-determined set of nodes.

5.3 Dynamic Source Routing

All of our previous examples use what is called *right* recursion, since the recursive use of *path* in the rule (NR2, DV2) appears to the right of the matching *link*. The query semantics do not change if we flip the order of *path* and *link* in the body of these rules, but the execution strategy does change. In fact, using *left recursion* as follows, we implement the Dynamic Source Routing (DSR) protocol [17]:

#include(NR1)
DSR1: path(S,D,P,C) :- **path(S,Z,P$_1$,C$_1$), link(Z,D,C$_2$),**
P = $f_concatPath$(P$_1$, link(Z,D,C$_2$)),
C = C$_1$ + C$_2$.
Query: path(S,D,P,C).

Rule NR1 produces new one-hop paths from existing link tuples as before. Rule DSR2 matches the destination fields of newly computed path tuples with the source fields of link tuples. This requires newly computed path tuples be shipped by their destination fields to find matching links, hence ensuring that each source node will recursively follow the links along all reachable paths. Here, the function $f_concatPath(P, L)$ returns a new path vector with L appended to P. These rules can also be used in combination with BPR1 and BPR2 to generate the best paths. By adding two extra rules not shown here, we can also express the logic for sending each path on the reverse path from the destination to the source node.

5.4 Link State

To further illustrate the flexibility of our approach, we consider a link-state protocol that moves route information around the network very differently from the best-path variants. The following *Link-State* query expresses the flooding of links to all nodes in the network:

LS1: floodLink(S,S,D,C,S) :- link(S,D,C)
LS2: floodLink(M,S,D,C,N) :- link(N,M,C$_1$),
floodLink(N,S,D,C,W), M $\neq$ W.
Query: floodLink(M,S,D,C,N)

$floodLink(\underline{M}, S, D, C, N)$ is a tuple storing information about *link(S,D,C)*. This tuple is flooded in the network starting from source node S. During the flooding process, node M is the current node it is flooded to, while node N is the node that forwarded this tuple to node M.

Rule LS1 generates a $floodLink$ tuple for every link at each node. Rule LS2 states that each node N that receives a $floodLink$ tuple recursively forwards the tuple to all neighbors M except the node W that it received the tuple from. Datalog tables are set-valued, meaning that duplicate tuples are not considered for computation twice. This ensures that no similar $floodLink$ tuple is forwarded twice.

Once all the links are available at each node, a local version of the *Best-Path* query in Section 5.1 is then executed locally using the $floodLink$ tuples to generate all the best paths.

5.5 Multicast

The examples we have given so far support protocols for unicast routing. Here, we demonstrate a more complex example, using Datalog to construct a multicast dissemination tree from a designated root node to multiple destination nodes that "subscribe" to the multicast group. The following *Source-Specific-Multicast* query sets up such a forwarding tree rooted at a source node a for group *gid*:

#include(NR1,NR2,BPR1,BPR2)
M1: joinMessage(I,N,P,S,G) :- joinGroup(N,S,G),
bestPath(N,S,P$_1$,C),
I = f_head(P$_1$), P = f_tail(P$_1$)
M2: joinMessage(I,J,P,S,G) :- joinMessage(J,K,P$_1$,S,G),
I = f_head(P$_1$),P = f_tail(P$_1$),
$f_isEmpty(P_1) = false$.
M3: forwardState(I,J,S,G) :- joinMessage(I,J,P,S,G).
Query: joinGroup(N,a,gid)

For simplicity of exposition, this query utilizes the *Best-Path* query (rules NR1, NR2, BPR1 and BPR2) to compute the all-pairs best paths. We will discuss query optimization techniques to reduce the communication overhead for small multicast groups in Section 7.2.

Each destination node n joins the group *gid* with source a by issuing the query $joinGroup(\underline{n}, a, gid)$. This results in the generation of the following derived tuples:

joinMessage(nodeID, prevNodeID, pathVector, source, gid). This tuple stores the multicast *join* message for group *gid*. It is sent by every destination node along its best path to the *source* address of the group. At each intermediate node with address *nodeID*, we keep track of *prevNodeID*, which is the address of the node that forwarded this tuple. *pathVector* is the remaining path that this message needs to traverse in order to reach the source node.

forwardState(nodeID, forwardNodeID, source, gid). This tuple represents source-specific state of the multicast dissemination tree at each intermediate node with address

nodeID. If a message from *source* of multicast group *gid* is received at *nodeID*, it is forwarded to *forwardNodeID*.

Rules M1 and M2 create the $joinMessage$ tuple at each participating destination node N, and forward this tuple along the best path to the source node S. Upon receiving a $joinMessage$ tuple, rule M3 allows each intermediate node I to set up the forwarding state using the $forwardState(\underline{I}, J, S, G)$ tuple. The predicate function $f_head(P)$ returns the next node in the path vector P, and $f_tail(P)$ returns the path vector P with the first node removed. $f_isEmpty(P)$ returns true if P is empty.

Instead of a *source-specific* tree, with minor modifications, we can construct *core-based trees* [9]. Here, each participating node sends a *join* message to a designated *core* node to build a *shared* tree rooted at the core. Messages are then unicast to the core, which disseminates it using the shared tree.

6. SECURITY ISSUES

Security is a key concern with any extensible system [24, 11]. In the network domain, this concern is best illustrated by Active Networks [14] which, at the extreme, allow routers to download and execute arbitrary code.

Our approach essentially proposes Datalog as a Domain Specific Language (DSL) [27] for programming the control plane of a network. DSLs typically provide security benefits by having restricted expressibility. Datalog is attractive in this respect, both because of its strong theoretical foundations, and its practical aspects. Queries written in the core[3] Datalog language have polynomial time and space complexities in the size of the input [5]. This property provides a natural bound on the resource consumption of Datalog queries.

However, many implementations of Datalog (including our own) augment the core language with various functions. Example of such functions are boolean predicates, arithmetic functions, and string or list manipulation logic (*e.g.*, $f_concatPath$, f_inPath, $f_isEmpty$, f_head and f_tail). With the addition of arbitrary functions, the time complexity of a Datalog program is no longer polynomial.

Fortunately, several powerful static tests have been developed to check for the termination of an augmented Datalog query on a given input [18]. In a nutshell, these tests identify recursive definitions in the query rules, and check whether these definitions terminate. Examples of recursive definitions that terminate are ones that evaluate monotonically increasing/decreasing predicates whose values are upper/lower bounded.

The queries that pass these checks are general enough to express a large class of routing protocols. Thus, our augmented Datalog language offers a good balance between expressiveness and safety. We note that all queries presented in this paper pass such termination tests, with the exception of the original *Network-Reachability* query in Section 3. This query has a rule NR2 that recurse infinitely to generate path tuples of monotonically increasing costs. However, with the addition of the boolean function $f_inPath(P_2, S) = false$ to prevent path cycles, the number of recursive calls are finite and hence the query is safe.

Datalog is a side-effect-free language which takes a set of stored tables as input, and produce a set of derived tables. In addition, the execution of the query is "sandboxed" within the query engine. These properties prevent the query from accessing arbitrary router state such as in-flight messages, and the router's operating system state. As a result, Datalog eliminates many of the risks usually associated with extensible systems.

Of course, there are many other security issues beyond the safety of the Datalog language. Two examples are denial-of-service attacks and compromised routers. These problems are orthogonal to network extensibility, and we do not address them in this paper.

[3]Such a "core" language does not contain predicates constructed using function symbols.

7. OPTIMIZATIONS

In this section, we explore connections between database query optimization techniques and routing protocols, with a focus on more efficient and realistic implementations of the examples above. In addition, we address techniques for work-sharing among a diverse set of queries, a new challenge that is not well-studied in either the database or networking literature.

7.1 Pruning Unnecessary Paths

A naïve execution of queries with aggregates such as *Best-Path* and *Distance-Vector* starts by enumerating all possible paths, and then selects among the result. This inefficiency can be avoided with a query optimization technique known as *aggregate selections* [25, 22]. Space constraints prevent a detailed discussion of this optimization, but we illustrate the idea with an example. In Figure 3, there are two different paths from node a to node d, but only the shorter of the two is required when computing shortest paths. By maintaining a "min-so-far" aggregate value for the current shortest path cost from node a to its destination nodes, we can selectively avoid sending path tuples to neighbors if we know they cannot be involved in the shortest path. In general, aggregate selections are useful when the running state of a monotonic *AGG* function (as in Section 5.1) can be used to prune communication. In addition, aggregate selections are necessary for the termination of some queries. For example, without aggregate selections, if paths with cycles are permitted, a query computing the shortest paths will run forever.

7.2 Subsets of Sources and Destinations

In Sections 5.2 and 5.5 we considered scenarios involving only a subset of nodes in the network. However, our examples so far – based on the core *Network Reachability* query of Section 3 – require all nodes to participate in the query plan. This leads to an unnecessary overhead when only a subset of nodes participate in the query as sources and/or destinations. Next, we discuss two techniques that alleviate this problem: *magic sets rewrite* and *left-right recursion rewrite*.

Magic Sets Rewrite: Consider the multicast construction in Section 5.5. Even when only a small number of nodes participate in the multicast group, the query will still compute the best paths between all pairs. To limit query computation to the relevant portion of the network, we use a query rewrite technique, called *magic sets rewriting* [10]. For example, if nodes b and c are the only nodes issuing the path query, the rewritten example is as follows:

MRR1: **magicSources(<u>D</u>) :- magicSources(<u>S</u>), link(<u>S</u>,D,C).**
MRR2: path(<u>S</u>,D,P,C) :- **magicSources(<u>S</u>)**, link(<u>S</u>,D,C),
P = $f_concatPath$(link(<u>S</u>,D,C), nil).
MRR3: path(<u>S</u>,D,P,C) :- **magicSources(<u>S</u>)**, link(<u>S</u>,Z,C_1),

path($\underline{Z}$,D,P_2,C_2), C = C_1 + C_2,
P = *f_concatPath*(link($\underline{S}$,Z,C_1),P_2).

MRR4: magicSources($\underline{b}$).

MRR5: magicSources($\underline{c}$).

Query: path($\underline{S}$,D,P,C).

The changes to rules NR1 and NR2 are represented in *bold*. Intuitively, the set of *magicSources* facts is used as a "filter" in the rules defining paths. After the rewrite, only nodes reachable from b and c participate in this query – the computation is restricted to just the relevant nodes in the network. The query can be further optimized by combining the common sub-rules at the beginning of MRR1, MRR2 and MRR3.

Left-Right Recursion Rewrite: The above rewritten query may provide little or no savings if the set of destinations is not constrained. Consider the example in Figure 3, where nodes b and c are the *only* source nodes. Even with magic sets, the computation of paths from these sources will require the computation of *all* paths sourced at *all* nodes reachable from b and c. To avoid these extra computations, we can rewrite the query using *left* recursion. To illustrate, the following *Best-Path-Pairs* query extends the previous query to perform (1) left recursion, and (2) magic sets query rewrite on *both* sources and destinations to generate best paths from all *magicSources* to *magicDsts* nodes:

BPP1: path(S,$\underline{D}$,P,C) :- magicSources($\underline{S}$), link($\underline{S}$,D,C),
P = *f_concatPath*(link($\underline{S}$,D,C), nil).

BPP2: path(S,$\underline{D}$,P,C) :- **path(S,$\underline{Z}$,P_1,C_1)**, **link($\underline{Z}$,D,C_2)**,
C = *f_compute*(C_1,C_2),
P = *f_concatPath*(P_1, link($\underline{Z}$,D,C_2)).

BPP3: pathDst(S,$\underline{D}$,P,C) :- **magicDsts($\underline{D}$)**, path(S,$\underline{D}$,P,C).

BPP4: bestPathCost(S,$\underline{D}$, AGG<C>) :- pathDst(S,$\underline{D}$,Z,C).

BPP5: bestPath($\underline{S}$,D,P,C) :- bestPathCost(S,$\underline{D}$,C), path(S,$\underline{D}$,P,C).

BPP6: magicSources($\underline{c}$).

BPP7: magicDsts($\underline{e}$).

Query: bestPath($\underline{S}$,D,P,C)

The above example computes only the required best path starting from the source node c to e. Rules BPP1 and BPP2 are used to compute the paths using *left* recursion starting from the *magicSources* nodes. Recall that the rules are *left* recursive because the recursive term *path* appears to the left of the matching *link*. As pointed out in Section 5, executing the query in a left recursive fashion bears close resemblance to dynamic source routing. Each source node computes new path tuples by recursively following the links along all reachable paths[4]. Filtering of the required destination nodes is done by matching *magicDsts* with the destination addresses of computed paths (rule BPP3). The best paths are then computed using rules BPP3 and BPP4, and sent back to the source nodes. By adding two extra rules not shown here, we can also express the logic for sending each best path on the reverse best path from the destination to the source node.

The drawback of this approach is that it does not allow computations along overlapping paths to be shared. For example, if *magicSources*($\underline{b}$) is added to the query, both nodes b and c must compute their paths to node e separately. In contrast, in the previous right-recursive query, both nodes b and c would be able to obtain that shared information from node d. In the following subsection, we will discuss a simple query rewrite for fixing this problem, yet retaining the benefits of left recursion.

[4]Note that the rules specifies that the computed path tuples are stored at the destination nodes instead of the source nodes as in the previous queries. This turns out to be the optimal tuple placement strategy that minimizes communication overhead for this query. While the decision on where to store derived facts is currently explicitly specified via the rules, we plan to explore letting a query optimizer decide the optimal placement automatically.

7.3 Multi-Query Sharing

As discussed in the introduction, we are interested in facilitating aggressive use of the routing infrastructure, in which a diverse set of route requests queries is executed concurrently in our system. A key requirement for scalability is the ability to share the query computation among a potentially large number of queries. A challenge for the query processor is in detecting sharing opportunities across the diverse set of queries. Detecting overlaps between Datalog queries (or database queries in general) is a difficult problem [12], and beyond the scope of this paper. However, we can leverage the fact that our routing queries are often simple variants of graph transitive closure computations. We are currently exploring the use of a more concise transitive closure language representation [6] that makes it easier to determine whether two queries are similar.

We first consider sharing among queries with *identical* rules, as might occur in a single-protocol scenario. If all nodes are running the same query, the optimal strategy is one based on right-recursion where each node directly utilizes path information sent by neighboring nodes. On the other hand, if only a small subset of nodes are issuing the same query, using left-recursion achieves lower message overhead as we will see in Section 9. In general, one would like an optimizer to automatically choose whether to use left or right recursion. This can be achieved using a query rewrite optimization. For example, the following *Best-Path-Pairs-Share* query replaces the original *left-recursion* rule BPP2 from the *Best-Path-Pairs* query with two rules BPPS1 and BPPS2:

#include(BPP1,BPP3,BPP4,BPP5)

BPPS1: path(S,$\underline{D}$,P,C) :- magicDst(D_3), path(S,$\underline{Z}$,P_1,C_1),
link($\underline{Z}$,D,C_2), ¬bestPathCache($\underline{Z}$,D_3,P_3,C_3),
C = *f_compute*(C_1,C_2),
P = *f_concatPath*(P_1, link($\underline{Z}$,D,C_2)).

BPPS2: path(S,$\underline{D}$,P,C) :- magicDst(D), path(S,$\underline{Z}$,P_1,C_1),
bestPathCache($\underline{Z}$,D,P_2,C_2),
C = *f_compute*(C_1,C_2),
P = *f_concatPath*(P_1,P_2).

Query: bestPath($\underline{S}$,D,P,C)

Rule BPPS1 specifies that in the absence of any cached[5] bestPath tuple, the original *left* recursion computation can be used. If a cached bestPath tuple generated previously is available, BPPS2 reuses the cached tuple instead. To illustrate, we revisit the example network in Figure 3. Consider the earlier case when two source nodes b and c are computing best paths to node e. If $bestPathCache(\underline{d}, e, [d, e], 1)$ is stored locally at d, it can be used by both nodes b and c using rule BPPS2. This avoids duplicate traversals of the path $d \rightarrow e$ and beyond. On the other hand, if this tuple is not present, the left recursion rule BPPS1 will be used instead.

Further sharing is achieved if the resulting path tuples are sent back via the reverse path to the source node to be reused by other queries. For example, when node a computes its best

[5]Our example here treats *bestPathCache* as a base table whose contents are not explicitly defined in the datalog rules. The logic for populating the cache is therefore not fully declarative. Addressing this issue is an intriguing direction for further research.

path to node e, the nodes on the reverse path (b and d) can cache information on the shortest path (and sub-paths) to node e, to be reused by subsequent queries.

Next, we consider the other mode of sharing, where queries have only partial similarity. We focus on the case where the rules are largely identical, with the exception of differences in function calls. For example, consider running two variants of the *Best-Path* query from Section 5.1, one that computes shortest paths, and another that computes max-flow paths. We can merge these into a single variant of the Best-Path query by simply tracking two running cost attributes (*e.g.*, path length and path capacity) and checking two aggregate selections (*e.g.*, min(path-length), max(capacity)). The merged query will share all path exploration across the queries. Aggregate selections continue to be applicable, but can only prune paths that satisfy *both* aggregate selections; pruning is effective when the selections are correlated.

8. STABILITY AND ROBUSTNESS

As discussed in Section 2, each query that is issued is accompanied by a specification of the desired lifetime (duration) of the computed route. During this period, changes in the network might result in some of the computed routes becoming stale. These can be caused by link failures, or changes in the link metrics when these metrics are used in route computation. Ideally, the query should rapidly recompute a new route, especially in the case of link failures.

One solution is to simply recompute the queries from scratch, either periodically or driven by the party that has issued the queries. However, recomputing the query from scratch is expensive, and if done only periodically, the time to react to failures is a half-period on average. The alternative approach we employ in this paper is to utilize long-running or *continuous* queries that incrementally recompute new results based on changes in the network. To ensure incremental recomputations, all intermediate state of each query is retained in the query processor until the query is no longer required. This intermediate state includes any shipped tuples used in join computation, and any intermediate derived tuples.

As we discussed in Section 2, each router is responsible for detecting changes to its local information or base tables and reporting these changes to its local query processor. These base tuple updates result in the addition of tuples into base tables, or the replacement of existing base tuples that have the same unique key as the update tuples. The continuous queries then utilize these updates and the intermediate state of the queries to incrementally recompute some of their derived tuples.

To illustrate, consider the *Network-Reachability* query in Section 3. Figure 4 shows a simple four node network where all four nodes are running the *Network-Reachability* query. Prior to the failure of node d, we assume that all paths between all pairs have been computed.

This query is executed at *all* nodes in the network, but for simplicity we focus on the tuples generated at nodes a and c. $p(\underline{S}, D, P, C)$ abbreviates $path(\underline{S}, D, P, C)$ and $l'(S, \underline{D}, C)$ refers to link tuples that are sent and cached at the destination nodes. We examine how $p(\underline{a}, d, [a, c, d], \infty)$ is created when node d fails:

1. When node d fails, neighbor c detects the failure and generates an updated base tuple $l(\underline{c}, d, \infty)$ locally. This replaces the previous tuple $l(\underline{c}, d, 1)$.

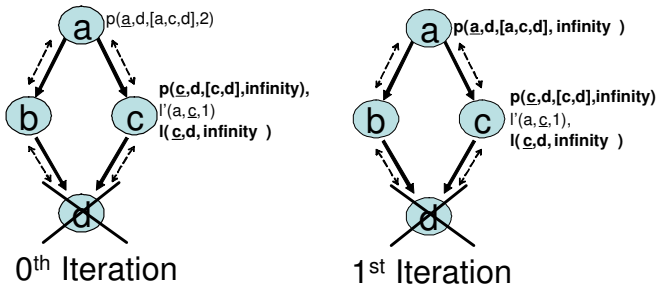

Figure 4: ***The figure shows the changes to the intermediate query states that led to the derivation of*** $p(\underline{a}, d, [a, c, d], \infty)$ ***when node*** **d** ***fails. For simplicity, we only show the states on nodes*** **a** ***and*** **c** ***necessary to show the derivation.***

2. All paths at node c that traverse through d are set to infinite costs[6]. For example, node c generates $p(\underline{c}, d, [c, d], \infty)$.
3. $p(\underline{c}, d, [c, d], \infty)$ is joined locally with $l'(a, \underline{c}, 1)$ to produce $p(\underline{a}, d, [a, c, d], \infty)$ which is sent to node a.

The failure is propagated hop-by-hop and in this example, since we compute the entire path vector and can check for potential cycles as described in Section 3, the time taken for any update to converge is proportional to the network diameter.

Updates to link costs are handled in a similar fashion, except that rather than setting the costs to infinity, they are recomputed based on the new link costs. The updated paths may trigger further computation. For example, when the cost of paths are changed, rules BPR1 and BPR2 of the *Best-Path* query will generate alternative best paths accordingly.

9. PERFORMANCE EVALUATION

To evaluate our solution, we have implemented a prototype system using PIER [2], a distributed relational query processor written in Java. Each node runs a PIER query engine, and maintains a neighbor table directly accessible by the PIER process. We have modified the PIER software to bypass the use of DHTs [7] and instead use explicit neighbor tables. A PIER process can contact only the PIER processes on neighbor nodes. Routing protocols expressed as queries can be issued directly to any PIER node, which then communicates with the neighbor PIER nodes to evaluate the queries.

We evaluate the system using a combination of simulations on transit-stub topologies (Section 9.1), and an actual deployment on PlanetLab (Section 9.2). Both the simulation and the actual implementation share the same code base.

Our evaluation suggests that our approach is feasible and that its expressiveness does not come at the expense of any significant degradation of scalability or performance. Our main results can be summarized as follows:

1. When all nodes issuing the same query, we show that the query execution has similar scalability properties as the traditional distance vector and path vector protocols.
2. When different set of nodes issuing different queries, the query optimization and work-sharing techniques are effective in reducing the communication overhead.
3. Our prototype deployment on PlanetLab shows that our system is able to react quickly to changes (either RTT fluctuations or churn) and find alternative paths.

[6]An additional rule *NR3:* $path(\underline{S}, D, P, \infty)$ *:-* $link(\underline{S}, W, C_1)$, $path(\underline{S}, D, P, C_2)$, $f_inPath(P, W) = true$, $C_1 = \infty$ is required.

9.1 Simulation Settings and Metrics

In our simulations, we run multiple PIER nodes on top of an event-driven network simulator that simulates bandwidth and latency bottlenecks. We generate transit-stub topologies using the GT-ITM topology generator [1]. The transit-stub topology consists of eight nodes per stub, three stubs per transit node, and four nodes per transit domain. We increase the number of nodes in the network by increasing the number of domains. The latency between transit nodes is set to 50 ms, the latency between a transit and a stub node is 10 ms, and the latency between any two nodes in the same stub is 2 ms. The capacity of each node is set to 10 Mbps; this is never a bottleneck in the query execution.

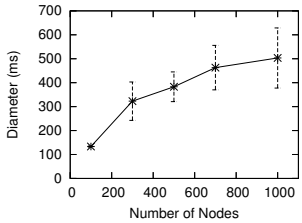

Figure 5: ***Network diameter vs Number of nodes.***

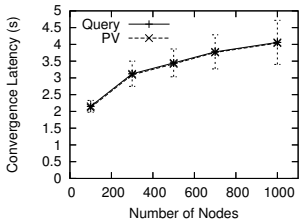

Figure 6: ***Convergence latency vs Number of nodes.***

We use five network sizes ranging from 100 to 1000. For each network size, we average all our experimental results over five runs. Figure 5 shows the diameter (latency of longest path) of the network as we increase the number of nodes, with the standard deviation error bars for each data point.

Our experiments consist of two different workloads. Our first workload (Section 9.1.1) involves a query that is being executed on all nodes, while our second workload (Section 9.1.2 and Section 9.1.3) involves a subset of nodes executing either the same or different query. We measure the performance of query execution using two metrics:

Convergence latency: time taken for the query execution to generate all the query results.

Per-node communication overhead (or simply **communication overhead**): the number of KB transfered on average per node during the query execution.

9.1.1 All-Pairs Shortest Paths

In our first experiment, we measure the performance of our system when all nodes are running the same query. We execute the *Best-Path* query as described in Section 5.1 to compute the shortest latency paths between all pairs of nodes. This query is disseminated from a random node, and each node that receives the query starts executing the query plan shown in Figure 2.

In our implementation, we use the *aggregate selections* optimization to avoid sending redundant path tuples (see Section 7.1). Each node collects tuples received from neighboring nodes, applies aggregate selections and computes new path tuples every 200 ms.

In Figure 6 shows the convergence latency for the *Best-Path* query (*Query* line) as the number of nodes increases. For validation, we compare the convergence latency against our own implementation of the path-vector protocol (PV line) for computing all-pairs shortest paths using the same simulation setup. We make two observations. First, as expected, the convergence latency for the *Best-Path* query is proportional to the network diameter, and converges in the same time compared to the path vector protocol. Second, the per-node communication overhead increases linearly with the number of nodes, as each node needs to compute the shortest path to every other node in the network. Both observations are consistent with the scalability properties of the traditional distance vector and path vector protocols, suggesting that our approach does not introduce any fundamental overheads.

9.1.2 Source/Destination Queries

Next, we study the effects of query optimization techniques on lowering the communication overhead when only a subset of paths are computed. Instead of computing all pairs, our workload consists of a collection of *Best-Path-Pairs* queries. Recall from Section 7.2 that *Best-Path-Pairs* is an optimized version of *Best-Path*, where some of the rules are rewritten using magic sets and left-right recursion optimization techniques to reduce the communication overhead. Queries are issued periodically every 15 sec. Each query computes the shortest path between a pair of nodes, and the result tuple is sent back on the reverse path to the source.

Figure 7 shows the per-node communication overhead, as the number of source/destination queries increases. In this experiment, we use a 200-node network. The *All Pairs* line represents our baseline, and shows the communication overhead for computing all pairs shortest paths. *Pair-NoShare* shows the communication overhead for running the *Best-Path-Pairs* query with no sharing across queries. When there are few queries, the communication overhead of *Pair-NoShare* is significantly lower than of *All Pairs*, as the later computes many paths which were never requested. However, as the number of queries increases, the communication overhead increases linearly, exceeding *All Pairs* after 130 queries.

Finally, *Pair-Share* shows the communication cost of executing the *Best-Path-Pairs-Share* query discussed in Section 7.3, which rewrites some of the rules in *Best-Path-Pairs* to facilitate work-sharing. *Pair-Share* clearly decreases the communication overhead of *Pair-NoShare*. As more queries are issued, the increase in the communication overhead diminishes, as each subsequent query has an increased chance of reusing previously generated results. However, as the number of queries increases beyond 240, *Pair-Share* becomes more expensive than *All Pairs*.

Figure 8 shows the results from the same experiment as Figure 7 as the number of source/destination queries increases to $39,800$ (199×200). The communication overhead for *Pair-Share* levels off at 605 KB. Here, we also examine the impact of limiting the choice of destination nodes on the effectiveness of sharing. This workload is illustrative of constructing a multicast tree which requires the shortest paths to a small set of nodes (see Section 5.5).

We compare *Pair-Share* against *Pair-Share (X% Dst)*, which limits the choice of destination nodes to $X\%$ of nodes. By limiting the choice of destination nodes to 20% (1%) of nodes, the communication overhead levels off at 119 KB (6) KB. This is because the smaller X, the higher the cache hit rate and the greater the opportunity for work-sharing. In fact, our experiments show that if fewer than 30% of nodes are chosen as destinations, executing *Best-Path-Pairs-Share* incurs lower message overhead than computing all-pairs shortest paths, irrespective of the number of queries.

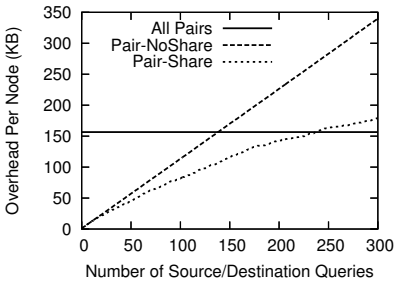

Figure 7: ***Average per-node communication costs for different query execution strategies (first*** 300 ***queries)***

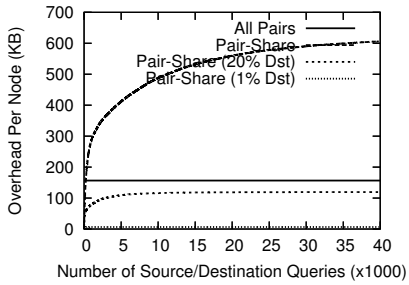

Figure 8: ***Average per-node communication costs for different query execution strategies (***$39,800$ ***queries)***

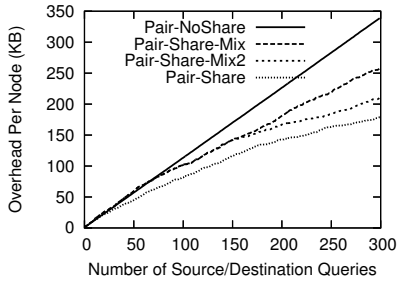

Figure 9: ***Average per-node communication costs for different query execution strategies (mixed query workload)***

9.1.3 Mixed Query Workload

So far, we have focused on query workloads consisting of identical queries between different source and destination nodes. In Figure 9, *Pair-Share-Mix* shows the communication overhead of running *Best-Path-Pairs-Share* queries on a *mixed* query workload. Each query computes the shortest path between a given source and destination node when 65% of the queries use the latency metric, while 5%, 10% and 20% of the queries use three other link metrics (with randomly generated values). As expected the communication overhead of the *Pair-Share-Mix* scenario lies between the *Pair-NoShare* and *Pair-Share* scenarios. This is because only queries that compute the same metric are likely to benefit from sharing.

Pair-Share-Mix2 shows the same mixed query workload as *Pair-Share-Mix*, with the exception that after 150 queries, the workload changes to one where all queries are using the latency metric. Once the workload changes, there are more work-sharing opportunities. Consequently, the communication overhead of each additional query for *Pair-Share-Mix2* is reduced compared to *Pair-Share-Mix*.

9.2 PlanetLab Experiments

To study the performance of our system in more realistic scenarios, we deploy our prototype on 72 PlanetLab nodes across three continents. The number of sites chosen for each experiment range from 30 to 35. These nodes form an overlay network, where each node hosts a PIER process.

In our experiments, we use the same metrics as before: the convergence latency, and per-node communication overhead. In all experiments, we use a single query load: the *Best-Path* query which is executed on all nodes and computes the shortest round-trip-times (RTT) paths for all pairs. We choose the RTT metric, instead of a more stable metric such as the hop-count, in order to stress our system under dynamic scenarios. We experiment with both one-time and continuous queries. With continuous queries, the link RTTs are periodically updated, and the query recomputes the path incrementally.

9.2.1 Preliminaries

In our experiments, we use three overlay topologies:

Sparse-Random: Each node randomly selects four other nodes as neighbors. This topology is a good stress case for our system. The links are usually much longer than in more realistic topologies (where there are more links between nearby nodes), which negatively affects the query convergence time.

Topology	AvgLinkRTT	AvgPathRTT
Sparse-Random	88 ms	185 ms
Dense-Random	88 ms	99 ms

Table 1: Average link RTTs (*AvgLinkRTT*) and shortest path RTTs (*AvgPathRTT*) for random topologies (experiments conducted on $2-5$ **Jan.**

Topology	AvgLinkRTT	AvgPathRTT
Dense-Random	106 ms	120 ms
Dense-UUNET	51 ms	89 ms

Table 2: Average link RTTs (*AvgLinkRTT*) and shortest path RTTs (*AvgPathRTT*) for dense topologies (experiments conducted on $17-19$ **Jan).**

Dense-Random: Same as above with eight neighbors.

Dense-UUNET: This topology is intended to represent a more realistic topology. Each node has an average degree of 8. Links between nodes at the same site are selected first. The remaining links are assigned as follows. We first divide the nodes based on their location into five coarse regions (North-America west/central/east, Europe and East Asia), and then select a mixture of intra-region and inter-region links that approximates the UUNET topology [4].

All our experiments were conducted during two time periods. The first set of experiments were conducted on the two random topologies during $2-5$ January, 2005. The second set of experiments were conducted on the *Dense-Random* and *Dense-UUNET* topologies during $17-19$ January, 2005. The load on PlanetLab during the second period was significantly higher than during the first period due to the approaching of conference deadlines. To account for load fluctuations, each experiment was conducted at least three times, and our experimental results were averaged across the multiple runs.

Tables 1 and 2 show the average RTTs of links (*AvgLinkRTT*), and the RTTs of computed paths (*AvgPathRTT*) when running the all-pairs shortest paths query. Since the links of *Sparse-Random* and *Dense-Random* are randomly chosen, the distribution of the link RTTs, and their average values are the same. However, notice that the average link RTT increases from 88 ms to 106 ms during the second set of experiments.

This is due to the heavier load on PlanetLab during the second period. As expected, the link RTT values in the case of *Dense-UUNET* are lower than *Dense-Random*, as *Dense-UUNET* contains more links between nearby nodes.

We also make two observations regarding the RTTs of the paths computed by our query. First, dense networks produce shorter paths due to their higher degree; the *AvgPathRTT* of *Sparse-Random* is twice as large as the *AvgPathRTT* of *Dense-Random*. Second, the *AvgPathRTT* for *Dense-UUNET* is lower compared to *Dense-Random*. This is because *Dense-UUNET* has more links between nearby nodes.

Since *Sparse-Random* and *Dense-Random* have long links, and since the RTTs of long links are more likely to fluctuate, these topologies will put a greater stress on our system than the *Dense-UUNET* topology. As a result, we focus on the two random topologies in our evaluation.

9.2.2 Query Execution

In this section, we evaluate the performance of a single query that computes the all-pairs shortest RTT paths on the two random topologies. The query uses the link RTTs as measured at the beginning of its execution.

Figure 10 shows the average path RTT (*AvgPathRTT*) versus time over the entire query execution period. *AvgPathRTT* at a given time t represents the average over the RTT values of all shortest paths computed by time t. A stable *AvgPathRTT* indicates that all shortest paths have been generated. At the start of query execution, *AvgPathRTT* increases steadily as paths are gradually discovered. After 20−30 sec, *AvgPathRTT* starts decreasing as shorter new discovered paths improve upon previously computed paths. 95% of the shortest paths are computed within 50 sec for *Sparse-Random* and 55 sec for *Dense-Random*. *AvgPathRTT* stabilizes after 73 sec and 82 sec, respectively.

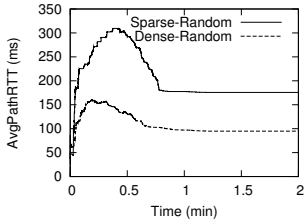

Figure 10: ***AvgPathRTT during query execution.***

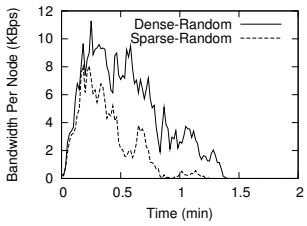

Figure 11: ***Query Bandwidth utilization.***

Figure 11 shows the per-node communication overhead for the same experiments. Initially, the per-node communication overhead increases steadily peaking at 8 KBps, and 11 KBps, respectively. As more shortest paths are discovered, the communication overhead starts to decrease steadily after 15 sec. The average communication overhead over the entire query execution period is 2.6 KBps for *Sparse-Random*, and 4.7 KBps for *Dense-Random*, respectively. While in the case of the *Dense-Random* topology the query has a higher overhead, it also produces paths with lower RTTs (see Table 1).

9.2.3 Path Adaptation

In this section, we consider the continuous version of the all-pairs shortest (RTT) paths query. After the initial query has computed all-pairs shortest paths, we begin to update the link RTT measurements every five minutes, and incrementally recompute new shortest paths as described in Section 8. In order to avoid ping congestion we spread the measurements uniformly across each five minute interval.

Topology	% Stable	Avg Change
Sparse-Random	33	3.8
Dense-Random	22	4.4
Sparse-Random (Smooth)	62	1.2
Dense-Random (Smooth)	42	1.6

Table 3: The computed path stability for random topologies with and without using RTT smoothing.

Figure 12 shows a representative experiment for the *Dense-Random* topology over a 50 minute interval during a busy period on PlanetLab. The *AvgPathRTT* value (solid line) follows the fluctuations of *AvgLinkRTT* (dotted line), which suggests that our system is able to recompute the shortest paths quickly as the underlying link RTTs change.

There is an inherent trade-off between quickly reacting to changes in the link RTTs, and the stability of the paths. When the query reacts to any changes in the link RTTs, the computed paths become less stable. This instability is quantified by the first two lines of Table 3, which shows results for the *Sparse-Random* and *Dense-Random* topologies. Only 33% and 22% of all paths remain unchanged after the initial query execution, and the shortest path between each pair of nodes changes on average 3.8, and 4.4, respectively. The denser topology is less stable, as there are more link updates per unit time.

The per-node communication overheads in steady state for *Sparse-Random* and *Dense-Random* are 586 Bps and 813 Bps, respectively. Despite the path instabilities, these overhead numbers represent only 22% and 17% of the overhead incurred by executing the complete query from scratch (see Section 9.2.2). Thus, recomputing the query results incrementally is both more efficient and faster than periodically reissuing the entire query.

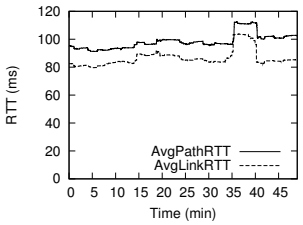

Figure 12: ***RTTs during query execution.***

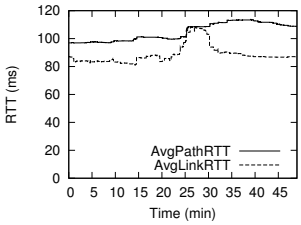

Figure 13: ***RTTs (with smoothing).***

To increase the stability of the computed paths, we use the classic Jacobson/Karels algorithm [20] to smooth the RTT values. For each link, we compute the mean deviation of the estimated RTT, and only send an update to the query processor if the latest estimated RTT exceeds the last reported by more than the mean standard deviation. We note that this estimation algorithm can be easily expressed in Datalog.

As shown in Table 3, smoothing the link RTTs is effective: the percentage of stable paths doubles, while the number of changes per path decreases by up to three times. Smoothing

also leads to a reduction of the overhead as fewer paths need to be recomputed. The average per-node bandwidth utilization in steady state for *Sparse-Random* and *Dense-Random* decrease to 175 Bps and 270 Bps, respectively. Finally, Figure 13 shows another 50 minute experimental run conducted roughly during the same period as the experiment in Figure 12 using smoothed RTT values. When using smoothing, *AvgPathRTT* is more stable despite similar RTT fluctuations.

9.2.4 Path Robustness under Churn

In this section, we study the performance of the continuous version of our query under churn. In addition, to our two metrics (per-node communication overhead and convergence latency), we add a third metric, *recovery time*. The recovery time of a path represents the time it takes the query to compute an alternate path from the moment it has detected the failure of the path. Note that the recovery time does *not* include the time to detect the failure. While detecting a failure is an important and non-trivial problem, we do not consider it in this paper.

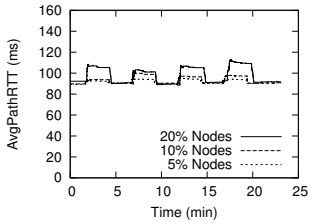

Figure 14: ***AvgPathRTT during query execution.***

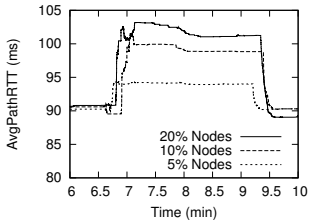

Figure 15: ***Close-up view of Figure 14.***

We conduct our experiments on the *Dense-Random* and *Dense-UUNET* topologies. In all experiments, we use smoothed link RTT values. We induce churn by alternately injecting *fail* and *join* events every 150 sec. At each *fail* event, a random set of nodes (chosen from either 5%, 10% or 20% of the nodes) experience fail-stop failures. This is followed by a *join* event where the previously failed nodes rejoin the network. The failure of each node will cause neighboring nodes to invalidate their neighbor entries. This generates *link* tuples with infinite costs and results in paths being invalidated as described in Section 8.

Figure 14 shows three experimental runs, for churn events consisting of 5%, 10% and 20% of all nodes of the *Dense-UUNET* topology. Each *fail* event corresponds to the sharp rise in *AvgPathRTT*, as longer routes are computed to avoid the failed nodes. Conversely, each subsequent *join* event causes the drop in *AvgPathRTT*, as new shorter routes are discovered. The higher the percentage of node failures, the greater the change in *AvgPathRTT* induced by a churn event.

Figure 15 shows the same results as Figure 14 over a four minute interval when a *fail* event is followed by a *join* event. Right after the failures, *AvgPathRTT* increases sharply as a large number of paths are invalidated and recomputed. However, *AvgPathRTT* decreases steadily as better paths are discovered and stabilizes.

Table 4 shows the breakdown of the recovery times for the experiments presented in Figure 14. A large fraction of paths recover almost instantly. The median recovery time is less than 1 sec, and the average recovery time is 2−2.2 sec. Only 1−3% of paths take more than 10 sec to recover. The average

% Nodes	Average (s)	% $\geq$ 10 s	Bandwidth
5	2	1	554 Bps
10	2	2	700 Bps
20	2.2	3	870 Bps

Table 4: Breakdown of the average recover time (s) of each path, the percentage of paths that require at least 10 s to recover, and average per node bandwidth (Bps) when there is 5%, 10% and 20% of node failures.

per-node communication overhead ranges from 554 Bps to 870 Bps, depending on the number of nodes participating in the churn event. In the worst case, the communication overhead is 18% of that required for recomputing the query.

In the case of the *Dense-Random* topology, our system demonstrates recovery times that are only slightly worse than for the *Dense-UUNET* topology. The median recovery time remains below 1 sec, as a large fraction of paths recover almost instantaneously. However, we observed that the average recovery time is higher (3.2−3.6 sec), and there is also an increase in the percentage of paths (3−6%) whose recovery time exceeds 10 sec. This increase is caused by the higher RTT values of the random topology links.

10. RELATED WORK

There have been many recent proposals for increasing the flexibility of routing in the context of the Internet. Proposed solutions include enabling end-hosts to choose paths at the AS level [28], separating routing from the forwarding infrastructure [19, 15], centralizing some of the routing decisions [15], and building extensible routers such as XORP [16]. Our proposal is mostly complementary to these efforts. The increased flexibility provided by a declarative interface can enhance the usability and programmability of these systems. Our proposal is also orthogonal to the separation of the control plane and the data plane. As discussed in Section 2, our system can be fully centralized, distributed or partially centralized.

Several type-safe languages have been proposed to improve the security and robustness of Active Networks. Two examples are PLAN-P [26] and SafetyNet [3]. Compared to these languages, Datalog is particularly attractive because of its strong theoretical foundations, the fact that it is a side-effect-free language sandboxed within a query engine, and its elegance in expressing routing protocols in a compact way. Unlike previous proposals, as a declarative query language, Datalog is also amenable to query optimization techniques from the database literature. Finally, we use Datalog exclusively for the control plane, and not for the data plane.

11. CONCLUSION

We propose *declarative routing*, which aims to strike a better balance between the extensibility of a routing infrastructure and its robustness. The basic idea of our solution is to express routing protocols using recursive query languages developed for deductive databases. Our solution can be viewed as an application of database techniques to the domain of networking, and is based on the key observation that recursive queries are a *natural* fit for expressing routing protocols.

We implemented a prototype system built on top of PIER, a distributed relational query processor. Using transit-stub

simulations and actual deployment on PlanetLab, we demonstrate that our system imposes no fundamental limits relative to traditional protocols, is amenable to query optimizations, and can efficiently sustain long-lived routes.

As future work, we will further explore the synergies between query optimization and network routing. We have identified a few well-known query optimization techniques and show how they can be used to generate efficient protocols. While these optimization techniques mimic well-known optimizations for routing protocols, it will be interesting to see how they can help inform new routing protocol designs. We intend to explore the use of an automatic query plan generator that not only optimizes each query individually, but also applies multi-query optimization techniques to automatically identify sharing opportunities among different queries.

We also plan to explore other uses of declarative queries in the network domain. These include a detailed study on expressing BGP inter-domain routing policies, and specifying declarative overlay networks.

12. ACKNOWLEDGEMENTS

The authors would like to thank Byung-Gon Chun, Tien Cheng Ee, Karthik Lakshminarayanan, Timothy Roscoe and Nina Taft for their thoughtful comments on early drafts of this paper. The authors would also like to thank Ryan Huebsch for support on the PIER code base. This research was funded by NSF grants IIS-0205657, IIS-0209108, 5710001486 and Intel Research Berkeley.

13. REFERENCES

[1] GT-ITM. http://www.cc.gatech.edu/projects/gtitm/.

[2] PIER. http://pier.cs.berkeley.edu.

[3] SafetyNet. http://www.cogs.susx.ac.uk/projects/safetynet/.

[4] WorldCom's Global UUNET Internet network. http://library.mobrien.com/manuals/mprm_group/uunet.jpg.

[5] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[6] R. Agrawal. Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries. In *IEEE Transactions on Software Engineering*, volume 14, 1988.

[7] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking Up Data in P2P Systems. *Communications of the ACM, Vol. 46, No. 2*, Feb. 2003.

[8] I. Balbin and K. Ramamohanarao. A Generalization of the Differential Approach to Recursive Query Evaluation. *Journal of Logic Programming, 4(3):259–262*, 1987.

[9] T. Ballardie, P. Francis, and J. Crowcroft. Core Based Trees (CBT): An Architecture for Scalable Inter-Domain Multicast Routing. In *SIGCOMM*, 2003.

[10] C. Beeri and R. Ramakrishnan. On the Power of Magic. In *PODS*, 1987.

[11] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *SOSP*, 1995.

[12] D. Calvanese, G. D. Giacomo, and M. Y. Vardi. Decidable Containment of Recursive Queries. In *ICDT*, 2003.

[13] W. F. Clocksin and C. S. Melish. *Programming in Prolog*. Springer-Verlag, 1987.

[14] D. Tennenhouse and J. Smith and W. Sincoskie and D. Wetherall and G. Minden. A Survey of Active Network Research. In *IEEE Communications Magazine*, 1997.

[15] N. Feamster, H. Balakrishnan, J. Rexford, A. Shaikh, and J. van der Merwe. The Case for Separating Routing From Routers. In *FDNA*, 2004.

[16] M. Handley, A. Ghosh, P. Radoslavov, O. Hodson, and E. Kohler. Designing IP Router Software. In *NSDI*, 2005.

[17] D. B. Johnson and D. A. Maltz. Dynamic Source Routing in Ad Hoc Wireless Networks. In *Mobile Computing*, volume 353. 1996.

[18] R. Krishnamurthy, R. Ramakrishnan, and O. Shmueli. A Framework for Testing Safety and Effective Computability. *J. Comput. Syst. Sci. 52(1)*, pages 100–124, 1996.

[19] T. V. Lakshman, T. Nandagopal, R. Ramjee, K. Sabnani, and T. Woo. The SoftRouter Architecture. In *HotNets-III*, 2004.

[20] L. Peterson and B. Davie. *Computer Networks: A Systems Approach*. Morgan-KaufMann, 2003.

[21] PlanetLab. http://www.planet-lab.org/.

[22] R. Ramakrishnan, K. A. Ross, D. Srivastava, and S. Sudarshan. Efficient Incremental Evaluation of Queries with Aggregation. In *SIGMOD*, 1992.

[23] R. Ramakrishnan and J. D. Ullman. A Survey of Research on Deductive Database Systems. *Journal of Logic Programming*, 23(2):125–149, 1993.

[24] M. Stonebraker. Inclusion of New Types in Relational Data Base Systems. In *ICDE*, 1986.

[25] S. Sudarshan and R. Ramakrishnan. Aggregation and Relevance in Deductive Databases. In *VLDB*, 1991.

[26] S. Thibault, C. Consel, and G. Muller. Safe and Efficient Active Network Programming. In *17th IEEE Symposium on Reliable Distributed Systems*, 1998.

[27] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Notices*, 35(6), 2000.

[28] X. Yang. NIRA: A New Internet Routing Architecture. In *Proceedings of FDNA-03*, 2003.