Automating Rendezvous and Proxy Selection

David Chiyuan Chu
Joseph M. Hellerstein

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-84
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-84.html

July 11, 2008

Automating Rendezvous and Proxy Selection

David Chu
EECS Department, UC Berkeley
davidchu@cs.berkeley.edu

Joseph M. Hellerstein
EECS Department, UC Berkeley
hellerstein@cs.berkeley.edu

***Abstract*—As increasingly diverse networks are interconnected, the combinations of environments and applications that will coexist will make custom engineering increasingly impractical. We investigate an approach which focuses on replacing custom engineering with automated optimization of protocol specifications. Interestingly, our optimizations, grounded in recursive query optimization, map smoothly to prototypical networking design points of rendezvous selection and proxy placement. Under two distinct networking settings, our prototype system can automatically choose program executions that are as much as three, and usually one order of magnitude better than original source programs.**

I. Introduction

Networks are growing increasingly diverse – or equivalently, diverse networks are increasingly inter-networked [5]. The causes of this diversity stem from both novel workload demands from above and new resource availability from below. From above, we are experiencing many new applications such as Video-On-Demand, telephony, and distributed sensing. From below, we are composing infrastructure from satellites, cellular networks, urban WiFi, and short-range radio like 802.15.4 and bluetooth.

Networking design methodology bears some resemblance to the early pre-relational database systems. The leading methodology for addressing physical and workload diversities in the network has been to engineer custom network program *implementations* one environment at a time. This approach may be difficult to scale; the combinations of environments and applications that will coexist in the networking space is poised to outstrip the ability to address each combination individually.

As an alternative approach, the lessons of data independence have been translated to the networking context, enabling high-level protocol *specifications* [26], [25], [24], [11]. Early work on this *declarative networking* approach focused largely on the ease of programming and conciseness of using recursive query languages to specify network protocols and architectures. In this work we take the lessons of database technology one step deeper, developing a database-inspired optimizer for network program specifications. This kind of automatic optimizer technology is timely in the networking environment, given increasingly diverse and varying workloads and resources.

Specifically, we focus on novel network-oriented adaptations of traditional Datalog optimizations that push selections in the presence of recursion. In the networking context, we show that these adaptations address prototypical networking rendezvous and proxy placement questions:

- Where should messages from communicating parties rendezvous?
- Who should hold the conversation state of an ongoing communication?
- Should applications send (application) data to routers, or conversely should routers send (routing) data to applications?

In addition to the utility of our optimizations, a conceptual contribution of our work is in exposing the congruency that naturally emerges between recursive query optimization and network provisioning.

Our algorithms are realized in `netopt`, our architecture for network optimization, which executes in three stages:

- *Analysis* identifies optimization opportunities from input programs. We show how to identify the three rendezvous and proxy selection opportunities above.
- *Rewriting* primes programs for optimization by transforming input programs to optimizable variants.
- *Decision Making* selects optimized configurations. The optimizer installs its chosen configuration by simply filling in tables initialized by *Rewriting* to list selected rendezvous and proxies.

To illustrate the utility of our network optimizations, we apply them to two distinct networking settings that have recently emerged: Content Distribution Networks (CDN) and Wireless Sensor Networks (WSN). In both scenarios, we demonstrate optimized programs significantly outperforming unoptimized original programs. In both emulated and simulated CDNs, performance improves by as much as three orders of magnitude. In emulated WSNs, gains are by as much as one order of magnitude. In both settings, `netopt` effectively identifies and executes better strategies.

Section I-A takes a closer look at our two chosen application scenarios. Section II introduces our distributed and recursive query language and offers initial attempts at network optimization. Section III discusses the main rendezvous optimization in detail. Section IV extends this to proxy placement optimization. Section V discusses their execution on our implementation platforms. Section VI reports on prototype implementation and deployment of our optimizations.

A. Two Networking Settings

This section takes a closer look at two networking settings: content distribution and wireless sensing. In each, we pay attention to aspects where manual rendezvous and proxy placement decisions have significantly impacted system design.

1) Content Distribution Networks: Website operators that wish to offer more responsive sites often turn to CDNs such as Akamai and Limelight [4], [23]. CDNs host third party content on their large server collections, ideally placing popular items close to interested consumers. Content placement in the CDN is an instance of rendezvous selection, which is also known as the NP-complete facilities location problem [12]). In the naive variant, the sink and source rendezvous at one endpoint. Yet, conceptually, rendezvous at any location in the distribution network is possible. Varying rendezvous impacts cost, and the optimization problem is akin to that in Publish-Subscribe and push-pull systems [16]: for minimal aggregate latency, what is the ideal rendezvous point for producers and consumers of each content item, subject to server storage constraints. We show how *Analysis* and *Rewriting* can automatically identify naive cases from program source and rewrite them to expose rendezvous flexibility. This permits *Decision Making* to assign lower cost rendezvous.

Each client that connects to a server allocates its own session state. As the number of clients increases, server responsiveness may suffer due to session state exhausting available server memory. To alleviate the problem, session state can be repackaged into messages for transportation between client and server, allowing the server to be stateless [34], [8]. Similarly, as a hybrid alternative, session state packaged in messages can be picked up from and dropped off at an intermediate proxy on the path connecting client and server. This proxy placement problem is complementary to rendezvous selection. We tackle it with automated techniques for exposing and optimizing proxy placement.

2) Wireless Sensor Networks: WSNs are embedded in and monitor the physical world over wide-ranging extents of space. As such, they present a fairly different context for automated network optimization. One predominant application class for WSNs is event detection. Often interchanging push for pull, event distribution and content distribution systems are known to be related problems. Therefore, optimizer-driven rendezvous selection applies to both WSN and CDN settings.

Recently, many common Internet services such as interactive login, remote debugging and point to point routing are being ported to sensornets. A challenge that arises repeatedly is that of configuring state allocation on storage-constrained platforms. Such state varies in form and use, from interactive login sessions to routing table entries. Should it reside at either endpoint, at intermediate proxies, or in packets? On the one hand, the general service designer is at a loss to properly provision for specific deployments. On the other hand, the end system deployer can not be expected to be intimately familiar with every packaged service. Unfortunately, this implies that neither is in the best position to optimize state allocation. We tackle this problem with automated techniques for exposing and optimizing proxy placement, much like those used for server session state proxy placement.

```
1 % Prepare for transmission
2 message(@Source, Source, Sink, Data) :-
3   produce(@Source, Data),
4   nexthop(@Source, Sink, Next).
5
6 % Route message to next hop parent
7 message(@Next, Source, Sink, Data) :-
8   message(@Current, Source, Sink, Data),
9   nexthop(@Current, Sink, Next).
10
11 % Receive if message is of interest
12 consume(@Sink, Data) :-
13   message(@Sink, Source, Sink, Data),
14   interest(@Sink, Data).
15
16 % What is consumed?
17 consume(@Sink, Data)?
```

Listing 1. Original `BasicProg`, message routing from source to sink with filtering by interest.

II. Example Program and Optimization

Throughout this work, the deductive database programming model is used as a means to demonstrate the concept of automated analysis, rewriting and optimization. The employed language, `netlog`, is convenient due to its immediate display of recursion, which we heavy utilize. `netlog` is a subset of OverLog [25] and is a dialect of Datalog.[1]

This section presents an example application, and an initial attempt to expose more rendezvous choices for the application. A main tool used throughout the optimizations, *network selection pushing*, is also introduced.

A. An Initial Program

Listing 1 introduces `BasicProg`, a `netlog` program that implements multi-hop message routing from sources to sinks with message filtering at sinks. The deductive database programming model employs *relations* and *deduction* as its basic constructs. Each relation consists of a set of tuples with the same number of attributes. Relations in `BasicProg` are $produce$, $consume$, $nexthop$, $message$ and $interest$. Deduction is expressed as a set of rules, each denoted by the symbol ":-" that indicates the existence of derived tuples based on the existence of other tuples. Each rule consists of a *body*, a set of conditions that appear to the right of the deduction symbol, and a *head*. The newly deduced data that appears to the left of the deduction symbol. Viewed operationally, this model is extremely simple: relations are best thought of as tables with columns in a database, tuples as table rows with values assigned to columns, and rules simply generate new table rows from existing table rows by equi-joining the tables in the body on the matching attribute variables.

Every tuple is stored at the network node indicated by its first attribute, the location specifier (denoted with the "@" symbol). This permits each node to hold a partition of each relation. For instance, a node's partition of a relation like $nexthop$ reflects its local routing table. When a rule involves location specifiers with differing variables, communication between nodes may occur to access necessary partitions.

In `BasicProg`, the $produce$ relation contains pairs of $Source$ and $Data$ attributes. When these tuples are joined

[1] Unlike `netlog`, OverLog distinguishes between events and stored tables.

against $nexthop$ tuples, initial $message$ tuples bound for $Sink$ are generated (lines 2-4). Joining produces an output tuple every time there exists tuples in the body that possess equal attribute values when the attribute names are the same. For example, a $message$ tuple's first attribute takes on the value $Source$ only when there exists a $produce$ and $nexthop$ tuple whose first attribute values match.

The data is routed by recursively defining the $message$ relation along the $nexthop$ relation. Intuitively, $message$ tuples are traversing the $nexthop$ routing tables (lines 7-9). Upon arrival at the sink, the $message$ tuple, if it matches any tuples in $interest$, generates $consume$ tuples at the destination (lines 12-14). The query (denoted with "?" symbol) indicates that a particular *queried relation* is made user-visible. Here, the query asks for the $consume$ queried relation (line 17).

B. Pushing Selections One-Hop

As an example, consider a two node network x and y represented by Extensional Database (EDB) $\mathcal{D}$:

```
produce(@y, foo). nexthop(@y, x, x). interest(@x, foo).
```

The EDB is materialized and is made of relations that are never in the head of any rule. Conversely, the Intensional Database (IDB) is made of the relations that occur in rule heads. The IDB corresponding to $\mathcal{D}$ is:

```
message(@y, y, x, foo). message(@x, y, x, foo). consume(@x, foo).
```

Here, $produce$ and $interest$ rendezvous at x via sending of a $message$ from y to x. Conceptually, this rendezvous could also take place at y as long as the query returns the same answer, $consume(@a, foo)$. To accomplish this, let $interest$ send its own "message" from x to y. We'll call it $message*$, and use it in the following rules:

```
message*(@Current, Current, Data) :-
  interest(@Current, Data).

message*(@Current, Sink, Data) :-
  message*(@Next, Sink, Data),
  nexthop(@Current, Sink, Next).

consume(@Sink, Data) :-
  produce(@Current, Data),
  message*(@Current, Sink, Data).
```

The first rule prepares $interest$ tuples for transmission. The second rule passes $message*$ backward along $nexthop$, and is similar to how $message$ was passed routed in Listing 1. The third rule derives $consume$. For the one-hop network, these rules produce the desired result of rendezvous at y, with the queried $consume$ at x.

C. Pushing Selections into the Network

As the network topology grows to multiple hops, we would like to add a bit more flexibility to this rewrite attempt. At the moment, we must choose between either endpoint. In a multi-hop network, rendezvous at any intermediary hop should be an option. We next provide some intuition on how network selection pushing generalizes to the multi-hop case. A program's network execution can be visualized with a *network derivation graph*. Figure 1a shows the network derivation graph for `BasicProg` over a four hop linear network with nodes x-y-z-w. Each network derivation graph node ρ_ξ represents a horizontal partition of relation ρ at location ξ (relation names are abbreviated by their first letter). A directed edge leads from derivation input to derivation output. For example, n_z represents the rows of $nexthop$ that are stored at location z, and the edge from p_w to m_w indicates that the program derives $message$ at w from $produce$ at node w. A node with a fan-in greater than one indicates a join among the node's children, as in the case of m_z and the join of m_w and n_w.

We can push selections to achieve a different network execution. Figure 1b shows the network derivation graph resulting from an initial selection push. Here, the join of $message$ and $interest$ is performed earlier, resulting in subsequent $message$ tuples already filtered by $interest$ (denoted $m-i$). Conceptually, the "pushing down" of $interest$ changes rendezvous of $message$ and $interest$ from x to z.

However, x and z are not neighbors in the underlying network topology (as indicated by the exclamation mark). Hence, they cannot communicate directly with each other and the partitions $interest_x$ and $message_z$ cannot directly join. In general, `netlog` programs require the following property for proper distributed execution.

Definition 2.1: A rule is **path-restricted** if all head and body relation partitions are located on the same host or neighboring hosts in the underlying network topology. A program is path-restricted if its rules are path-restricted.

We assume that input programs are path-restricted, and we would like to maintain the property for any rewritten programs. Figure 1c suggests an alternate join rearrangement that is path-restricted for $interest_x$. It is roughly the result of combining Listing 1 with the rules in Section II-B. $produce_w$ is converted to $message$ and travels from w to z, $interest_x$ is converted to $message*$ and travels from x to y to z. This leaves $message*_z$ and $message_z$ ready to join at z.

However, the derivation of $consume_x$ involves z and x that are not neighbors. This time we choose to "package up" $consume_x$ as $message**$ and send it along the network topology via a path we already know about from z to x. Figure 1d shows the fully path-restricted network derivation graph with rendezvous at z. Yet, this is just one possible rendezvous choice. "Meet-in-the-Middle" *MiM Rewrite* transforms input programs to expose many possible rendezvous choices.

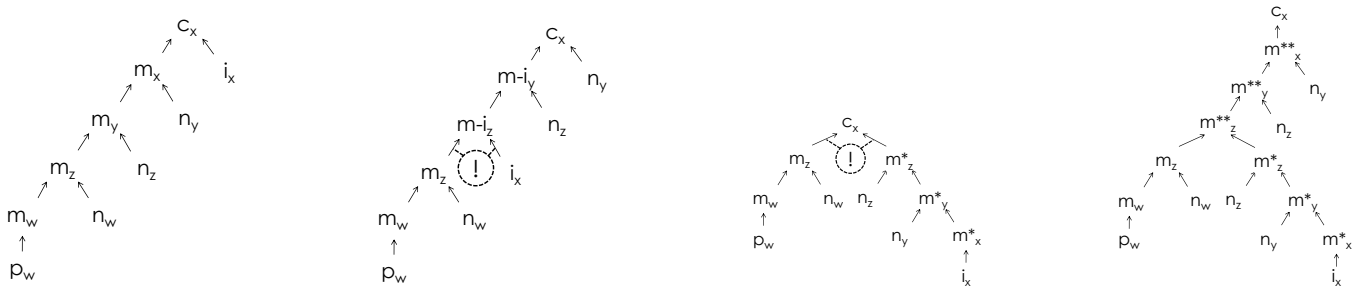
III. MiM Rewrite

This section first sets forth the correctness criteria of any `netopt` optimization, next describes the MiM Rewrite procedure precisely in terms of its analysis and rewrite phases, and lastly proves MiM Rewrite transformations are correct.

Any `netopt` optimization must preserve the intent of the original program. The intent is captured by the query.

Definition 3.1: Two programs P_1, P_2 are **query equivalent** if, given any EDB, the contents of their queried relations are equivalent.

Definition 3.2: A rewriter $R : P_1 \rightarrow P_2$ is **query preserving** if for all programs P_1, P_2 is query equivalent to P_1.



(a) Original (b) Initial selection push (c) Path-restricted $interest$ (d) Fully path-restricted

Fig. 1. Alternative executions of `BasicProg`. Exclamation marks indicate neighboring hosts are not connected in the network topology.

This section builds up to the establishment of the following.

Theorem 3.3: The MiM Rewrite is query preserving and path-restricted.

A. Analysis

Analysis identifies certain rules and relations as rewrite components. We first introduce some terminology from classic work in the deductive database literature [39].

Definition 3.4: A **rule-goal graph** contains one relation-node for each relation and one rule-node for each rule. A directed edge leads from rule-node $\mathcal{R}$ to relation-node a if the head of rule $\mathcal{R}$ is relation a. A directed edge leads from relation-node a to rule-node $\mathcal{R}$ if relation a is in the body of rule $\mathcal{R}$.

Definition 3.5: A rule with head relation a is a **linearly recursive rule** (LR rule) if a appears exactly once in the body. It is an **initializer rule** if a does not appear in the body.

Definition 3.6: A program is a **linearly recursive program** (LR program) if every rule with head a is (1) either an initializer rule or LR rule, and (2) for every relation b in the body, $b \neq a$, relation-node b in the rule-goal graph is not reachable from relation-node a.

Definition 3.7: A relation a is a **LR relation** if a is the head of a LR rule. The other relations in the body of the LR rule are **base relations**.

We can consider scenarios in which the LR rule body contains only one base relation. This does not sacrifice generality since it is straightforward to construct a single base relation by joining multiple base relations. Furthermore, our focus on networking programs leads us to consider the following type of LR rule.

$$\mathcal{R}_1 \;\; \mathsf{a}(@\mathsf{b}_i, \mathsf{d}_1, \ldots, \mathsf{d}_{Na-1}) \;:-\; \mathsf{a}(@\mathsf{a}_1, \ldots, \mathsf{a}_{Na}),\; \mathsf{b}(@\mathsf{b}_1, \ldots, \mathsf{b}_{Nb}).$$

Values of b_i are "injected" into the location specifier of a tuples upon every recursion, which means the partition of the head a is potentially different from the partition of the body a upon every recursion. Hence, we can interpret the base relation as defining a network for the LR relation to hop along. For `BasicProg`, $message$ is the only LR relation. $nexthop$ is a base relation of $message$. Both LR and base relation identification can be accomplished by traversing the rule-goal graph.

Note each d_i can correspond to any a_i or b_i to capture additional projections. Furthermore, b_i's can correspond to a_i's to capture joins.

Lastly, we are only interested in recursive relations that can (possibly indirectly) derive the queried relation because only they can impact query equivalency.

Definition 3.8: Given queried relation c and LR relation a, a rule $\mathcal{R}$ is an **answer rule** if (1) a is in the body of $\mathcal{R}$ but not the head, and (2) in a rule-goal graph traversal, rule-node $\mathcal{R}$ can reach relation-node c.

Given a program and queried relation, *Analysis* identifies LR and base relations, and LR, initializer and answer rules.

B. Rewriting

Using the rules and relations identified in *Analysis*, *Rewriting* invokes the MiM Algorithm. The MiM Algorithm transforms LR program P to a query equivalent program P_{MiM}. The advantage of P_{MiM} over P is that its rendezvous can be tuned by *Decision Making* by filling in tuples for a special $rendezvous$ relation.

A preliminary procedure of MiM Algorithm canonicalizes the input program. First, recursive relation a is renamed a_ans in every answer rule for a. Second, for each rule, each variable is renamed to a unique variable name that does not appear elsewhere in the program. Third, the queried relation and EDBs generate a *binding* for each recursive relation a. Intuitively, bindings are values for attributes of a that (1) we already know since they are join keys with materialized EDB relations and (2) are useful since the join happens in an answer rule. Operationally, this is done by employing the classic database Sideways Information Passing (SIP) algorithm [2]. The input to SIP is a queried relation c and program P. The output is a a sequence of "b"s (bound) and "f"s (free) for each a, called *binding list* α. For sake of space, we describe the algorithm only as it applies to the following type of answer rule where c is the head and e is in the EDB.

$$\mathcal{R}_0 \;\; \mathsf{c}(@\mathsf{c}_1, \ldots, \mathsf{c}_{Nc}) \;:-\; \mathsf{e}(@\mathsf{e}_1, \ldots, \mathsf{e}_{Ne}),\; \mathsf{a}(@\mathsf{a}_1, \ldots, \mathsf{a}_{Na}).$$

In this most basic yet common case, the binding list α is assigned as follows: α_i is "b" if a_i joins with some e_j.

INPUT: A LR input program P with a binding list α for each recursive relation a. Recursive rules for a take the form:

$\mathcal{R}_1$ a($\overline{a1}$) :- a($\overline{a2}$), b($\bar{b}$).

OUTPUT: An output program P_{MiM} having all the rules of P, with additional EDB relation r (rendezvous) and with each rule $\mathcal{R}_1$ replaced by rules $\mathcal{R}_{1.1}$, $\mathcal{R}_2$, $\mathcal{R}_{3.1}$, $\mathcal{R}_{4.2}$, $\mathcal{R}_5$ and $\mathcal{R}_6$ as defined below.

PROCEDURE:

1. *Invert recursion order.* Generate P_{inv}, a version of P that processes derivations via "pull" rather than "push". The rules of P_{inv} are the rules of P with each rule $\mathcal{R}_1$ replaced by three rules:

$\mathcal{R}_2$ a$*$($\overline{a0_b}$,$\overline{a0_b}$) :- ... *% answer rule dependent, refer to text*
$\mathcal{R}_3$ a$*$($\overline{a2_b}$,$\overline{a0_b}$) :- a$*$($\overline{a1_b}$,$\overline{a0_b}$), b($\bar{b}$).
$\mathcal{R}_4$ a_ans($\overline{a0_b}$,$\overline{a3_f}$) :- a$*$($\overline{a3_b}$,$\overline{a0_b}$), a($\overline{a3}$).
with $\overline{a0_b}$ = ... *% answer rule dependent, refer to text*
and $\overline{a3} = unique(Na)$.

2. *Hybridize recursion order.* Generate P_{hyb} by combining P_{inv} and P. In addition, add rendezvous relation r and modify selected rules to:

a. Limit derivations of the queried relation to the rendezvous point. Replace $\mathcal{R}_4$ with:

$\mathcal{R}_{4.1}$ a_ans($\overline{a0_b}$,$\overline{a3_f}$) :- a$*$($\overline{a3_b}$,$\overline{a0_b}$), a($\overline{a3}$), r($\overline{a3_b}$).

b. Limit "push" execution to before the rendezvous and limit "pull" execution to after the rendezvous. Replace $\mathcal{R}_1$ and $\mathcal{R}_3$ with:

$\mathcal{R}_{1.1}$ a($\overline{a1}$) :- a($\overline{a2}$), b($\bar{b}$), −r($\overline{a2_b}$).
$\mathcal{R}_{3.1}$ a$*$($\overline{a2_b}$,$\overline{a0_b}$) :- a$*$($\overline{a1_b}$,$\overline{a0_b}$), b($\bar{b}$), −r($\overline{a1_b}$).

3. *Localize for network processing.* Generate P_{MiM} by modifying P_{hyb} to ensure network topology path restrictions. This enables correct distributed execution. Replace rules $\mathcal{R}_{4.1}$ with:

$\mathcal{R}_{4.2}$ a$**$($\overline{a3_b}$,$\overline{a0_b}$,$\overline{a3_f}$) :- a$*$($\overline{a3_b}$,$\overline{a0_b}$),a($\overline{a3}$),r($\overline{a3_b}$).
$\mathcal{R}_5$ a$**$($\overline{a1_b}$,$\overline{a0_b}$,$\overline{a3_f}$) :- a$**$($\overline{a2_b}$,$\overline{a0_b}$,$\overline{a3_f}$), b($\bar{b}$).
$\mathcal{R}_6$ a_ans($\overline{a0_b}$,$\overline{a3_f}$) :- a$**$($\overline{a0_b}$,$\overline{a0_b}$,$\overline{a3_f}$).

Listing 2. MiM Algorithm

Otherwise α_i is "f".[2] Furthermore, we can safely assume that α is a sequence of "b"s followed by a sequence of "f"s. This may require some trivial variable reordering.

With these preliminaries settled, core MiM Algorithm in Listing 2 is invoked. The shorthand notation it uses allows us to present MiM Algorithm compactly as a series of rule manipulations and variable list rearrangements. A term with a bar ("$^-$") represents a list of variables. The first letter of the term indicates the size of the list. For example, $\overline{a1}$ represents a variable list of size Na. Each variable list comes from either (1) the input program P or (2) the function $unique$. Lastly, a term may have a subscript "b" or "f" to represent the application of the function $boundlist$ or $freelist$ respectively. For example, $\overline{a1_b} = boundlist(\overline{a1})$. In such case, the length of $\overline{a1_b}$ may be less than that of $\overline{a1}$.

$unique$ takes as input a list size and returns as output a list of distinct variables that do not appear anywhere else in any rule. $boundlist$ takes as input a variable list for a and returns the prefix of the input for which α is "b". Conversely, $freelist$ returns the suffix of the input for which α is "f".

MiM Algorithm generates new rules and introduces new relations a_ans, $a*$ and $a**$. In networking settings, tuples of a, $a*$ and $a**$ can be thought of as messages. Each message consists of a message header (some prefix of attributes) and message payload (remaining suffix of attributes). The header may change on every recursion but the payload does not.

MiM Algorithm consists of three main steps traced by Figure 2. Figure 2a shows the input program as an abstracted network derivation graph in which messages of a flow from source to sink. After arriving at the sink, a generates a_ans, which participates in answer rules (not shown). More precisely, sources are locations where initializer rules generate a, and sinks are locations where answer rules use a.

Step 1 inverts the recursive order of the original program. Its objective is the same as to that of the well-known Magic Sets algorithm [7]: pushing selection past recursion. This is done by constructing $a*$ to recurse backward from sink to source (Figure 2b). Pushing down selections can be thought of as a "pull" execution vs. the original "push" execution.

In Step 1 of Listing 2, $\mathcal{R}_2$ is underspecified and we complete its specification here. Recall that given a queried relation c, α tells us that some attributes of a are already bound to specific values because these attributes are join keys with EDBs in answer rules. Much like a semi-join, these join keys are simply copied from the EDB relations to make $a*$, a superset of a. In the case of example $\mathcal{R}_0$, $\mathcal{R}_2$ takes the form:

a$*$($\overline{a0_b}$,$\overline{a0_b}$) :- e($\bar{e}$). with $\overline{a0}$ = @a$_1$,...,a$_{Na}$ of $\mathcal{R}_0$

Note that two copies of the join keys are made. The first copy is like a message header that may need to go through some number of recursive modifications to find its join partners. The latter "pristine copy" is like a message payload with a return address inside, used to remember the original join keys for use with the answer rules.

Step 2 hybridizes the recursion order by combining push and pull execution to "meet-in-the-middle". It further introduces the EDB relation r whose tuples indicate the precise rendezvous meeting point between push and pull. While a traverses forward from source and $a*$ traverses backward from sink, both stop at the rendezvous to derive a_ans (Figure 2c). Whereas P_{inv} pushes selection past recursion, P_{hyb} pushes selection into a tunable middle point in the recursion.

Step 3 localizes the program for network processing by ensuring that network topology paths are respected. Essentially, a_ans is additionally packaged as a payload in another message, $a**$, and sent from rendezvous to sink. Upon reaching the sink, a_ans is unpackaged and can be used in answer rules, just as in the original program.

Steps 1 and 2 are applicable to any LR Datalog program. Step 3 is necessary for `netlog` programs that are expected to run on networks of nodes. Additional restrictions covered in Section III-D apply to input programs for MiM Algorithm. Before discussing these, we first present an example application of MiM Algorithm.

C. Example Application of MiM Algorithm

Listing 3 shows the result of a full application of MiM Algorithm on `BasicProg`. In the rewritten `BasicProg`

[2]In more complex cases, *e.g.,* where c and a do not participate in the same rule, SIP propagates bindings transitively across rules [2]. For our purposes, it is sufficient that SIP can always generate a single unique α for each a.

(a) Input program P (b) After Step 1: P_{inv} (c) After Step 2: P_{hyb} (d) After Step 3: P_{MiM}

Fig. 2. Steps of MiM Algorithm

```
1 % Prepare for transmission
2 message(@Source, Source, Sink, Data) :-
3   produce(@Source, Data),
4   nexthop(@Source, Sink, Next).
5
6 % Route message to next hop parent until rendezvous
7 message(@Next, Source, Sink, Data) :-
8   message(@Current, Source, Sink, Data),
9   nexthop(@Current, Sink, Next),
10  -rendezvous(@Current, Sink, Data).
11
12 % Route interest back along next hop until rendezvous
13 message*(@Current, Current, Data) :-
14   interest(@Current, Data).
15 message*(@Current, Orig, Data) :-
16   nexthop(@Current, Sink, Next),
17   message*(@Next, Orig, Data),
18   -rendezvous(@Next, Sink, Data).
19
20 % At rendezvous, join message and interest and send to Sink
21 message**(@Current, Sink, Data) :-
22   message(@Current, Src, Sink, Data),
23   message*(@Current, Sink, Data),
24   rendezvous(@Current, Sink, Data).
25 message**(@Next, Sink, Data) :-
26   message**(@Current, Sink, Data),
27   nexthop(@Current, Sink, Next).
28 consume(@Sink, Data) :-
29   message**(@Sink, Sink, Data).
30
31 % What is consumed?
32 consume(@Sink, Data)?
```

Listing 3. Rewritten `BasicProg`, message and interest meet in the middle.

of Listing 3, the precise rendezvous location is chosen by simply filling in the $rendezvous$ relation *e.g.,* with $rendezvous(@b, a, foo)$. The original recursion of $message$ along $nexthop$ is amended to include a negated term, $-rendezvous$ which modifies the interpretation of message routing to be: "Route $message$ along $nexthop$ until encountering $rendezvous$" (line 10). A similar negated term is applied to the routing back of $interest$ (line 18). Additionally, MiM Rewrite amends `BasicProg` to deliver $consume$ tuples in a multi-hop fashion to the $Sink$ (lines 21-29) according to network path restrictions mentioned earlier.

Note that we have not specified nor constrained the tuples in the $rendezvous$ relation. The decision of what to put there will be the task of *Decision Making*, discussed in Section V.

D. MiM Rewrite Correctness

This section proves Theorem 3.3 mentioned at the start of this section, and its structure mirrors Listing 2.

[1. Inverting Recursion Order] We need the following constraint to show query equivalency of P_{inv} and P.

Constraint 1 (Free Variable Constraint): For each recursive rule having head a, the free variables of a in the head must be the same as the free variables of a in the body.

This constraint says that the free variables of a are carried along unmodified from source to sink. Conversely, the bound variables of a may change upon every recursion. By analogy to networking, free variables are message payloads and bound variables are message routing headers. Provided this constraint, we borrow results from [39] (specifically Theorem 15.1) to claim that:

Lemma 3.9: P and P_{inv} are query equivalent.

Informally, we have already discussed how the construction of $\mathcal{R}_2$ causes $a*$ to be filled with an initial superset of join key values for a starting at sink. $\mathcal{R}_3$ takes the current set and asks what prior set is necessary to derive the current set, much like a depth first search from sink to source. Recall also that $\mathcal{R}_2$ and $\mathcal{R}_3$ store a "pristine copy" of the initial set, $\overline{a0_b}$, in the latter half of $a*$. Like a semi-join, (a subset of) $a*$ may eventually join with a at the source by $\mathcal{R}_4$.[3] If this happens, since the free variables $\overline{a3_f}$ do not change with recursion (due to the constraint), they can be copied over directly from a to a_ans. Similarly, $a*$ copies its pristine copy $\overline{a0_b}$ to a_ans. The result is a_ans, which can now be used by any answer rule since free variables for a_ans have now been assigned values. The proof of Lemma 3.9 is by induction on the length of the path from source to sink.

[2. Hybridizing Recursion Order] We can simultaneously enact push and pull processing by combining P and P_{inv}. However, this naive hybrid program causes many unnecessary derivations of identical a, $a*$ and a_ans tuples because neither push nor pull can detect when it has started duplicating the other's work; a goes completely from source to sink, $a*$ goes completely from sink to source, and at each point along the path, the same a_ans tuples are (re)derived. Limiting redundant derivations is not necessary for correctness but is preferable especially when redundant derivations lead to network communications overhead.

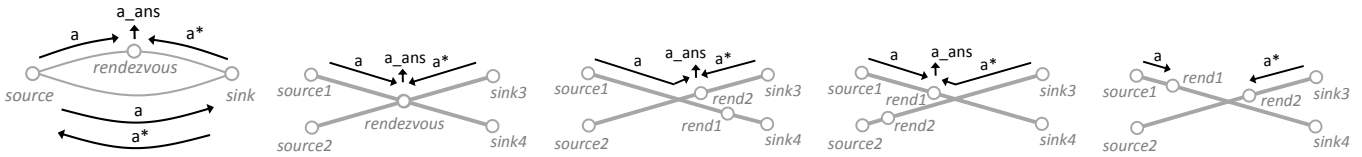
First, to limit redundant a_ans derivations, we name a particular rendezvous in a special rendezvous relation r. In Step 2.a, *Rewriting* adds r to the body of $\mathcal{R}_4$, creating $\mathcal{R}_{4.1}$. *Decision Making* populates the tuples of the r relation. It may choose any r as long as it obeys the following constraint.

Constraint 2 (Selection Constraint): Any source and sink pair that share at least one path must have at least one rendezvous on one of the shared paths.

Figure 3a shows an example where this constraint is respected. There are two paths from source to sink and at least one path, the top one, includes a rendezvous. No derivations of a_ans occurs on the bottom path due to failure to rendezvous *i.e.,* no entry of r lies on the bottom path.

Second, to limit redundant a and $a*$ derivations, we add r

[3]Recall that initializer rules can also generate a.



(a) Selection Constraint OK (b) Branching Constraint OK (c) Branching Constraint OK (d) Branching Constraint OK (e) Branching Constraint BAD

Fig. 3. MiM Algorithm constrains the location of rendezvous.

to the body of $\mathcal{R}_1$ and $\mathcal{R}_3$, creating $\mathcal{R}_{1.1}$ and $\mathcal{R}_{3.1}$. Here, r is negated, which means a and $a*$ traverse from source and sink respectively until encountering a rendezvous, but no further. To maintain correctness, *Decision Making* respects the following constraint in addition to Constraint 2.

Constraint 3 (Branching Constraint): Any single path between source and sink can have at most one rendezvous.

Figures 3b-3e show three examples respecting this constraint and one example violating this constraint. The setting is one in which $source1$ and $source2$ both have paths to $sink3$ and $sink4$. Such path "branches" at an intersection are analogous to network multicasts/one-to-many communication. Figure 3e is a violation because a from $source1$ and $a*$ from $sink3$ do not rendezvous at the same place. Provided Constraint 2 and 3, we have query equivalency after Step 2.

Lemma 3.10: P and P_{hyb} are query equivalent.

Proof: First consider P_{hyb} without the negated r terms. Since r's $\overline{a3_b}$ is a subset of a's $\overline{a3}$, $\mathcal{R}_{4.1}$ cannot derive a_ans tuples that were not derived by $\mathcal{R}_4$ (soundness). For completeness, proceed by induction on the choice of rendezvous. For the base case, choose the sink as the rendezvous, in which case P_{hyb} simplifies to P. For the induction, we can assume that query equivalency holds were we to pick the rendezvous one step closer to the sink at point $i+1$ rather than the current rendezvous choice at point i. By Lemma 3.9, LR relation a at $i+1$ implies a at i. Apply $\mathcal{R}_3$ to $a*$ at $i+1$ to derive $a*$ at i. Finally apply $\mathcal{R}_{4.1}$ to a, $a*$ and r (all at i) to derive a_ans. Constraint 2 ensures there is at least some rendezvous between source to sink for the inductive step.

Next consider P_{hyb} with the negated r terms. The addition of a negated term cannot make more derivations than without the negated term (soundness). Constraint 3 ensures that one path's rendezvous choice does not "block" rendezvous along another path, as seen in Figure 3e (completeness). ■

[3. Restricting Derivations to Network Paths] $\mathcal{R}_{4.1}$ is not a path-restricted rule. From the perspective of Figure 2c, a_ans is derived at the rendezvous but answer rules are expecting to use it at the sink. This is not an issue in a centralized Datalog execution. However, since the location specifier horizontally partitions relations in `netlog`, P_{hyb} needs path-restricting.

Path-Restricting Subprocedure modifies $\mathcal{R}_{4.1}$ to ensure that head and body location specifiers are the same or are neighbors. It accomplishes this by providing hop-by-hop delivery of a_ans from body location to head location.

Specifically, Path-Restricting Subprocedure prepends "message header" attributes to a_ans as new relation $a**$ (akin to encapsulation used in network tunneling). The message header is copied directly from a since it already had the appropriate header to get to the sink. This converts $\mathcal{R}_{4.1}$ to $\mathcal{R}_{4.2}$. Next, Path-Restricting Subprocedure constructs $\mathcal{R}_5$ to let $a**$ mimic a's multi-hop delivery. Lastly, $\mathcal{R}_6$ unpackages $a**$ into a_ans upon detecting that the outer message header is the same as the inner message header.

Finally, we claim Theorem 3.3.

Proof: From Lemma 3.10 we already know P and P_{hyb} are query equivalent. To see that P and P_{MiM} are also query equivalent, observe that the message payload in $a**$ representing a_ans is always copied and never modified. To see that P_{MiM} is path-restricted (provided P is path-restricted), proceed by induction on the choice of rendezvous, starting from the sink. To verify the base case, use the assumption that the input program is already path restricted. ■

Path-Restricting Subprocedure is a generalization of link-restricted rewrites first proposed in [24] from the one-hop/non-recursive rules to multi-hop networks/recursive rules. We shall reuse it in the forthcoming rewrites.

IV. Additional Rewrites

This section discusses two rewrites that address proxy placement, and both extend naturally from MiM Rewrite. Interestingly, in networking, proxy placement and rendezvous selection are typically not seen as related, but the connection is clear through the lens of query optimization. This section also discusses a third rewrite that increases rendezvous flexibility by enabling off-path redirection.

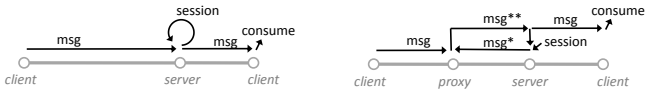
A. Session Proxies

Many protocols and services maintain per-conversation session state at endpoints. For example, web cookies allow a web service to bind client requests to corresponding *session state*, *i.e.,*, data about the client connection kept on the server (stateful server). However, if a server gets many simultaneous connections and exhausts its own storage for session state, it may prefer to offload the state to proxies (proxied server) or even to shuttle the session state back and forth with the client in each packet (stateless server).[4] This conversion of session state is applicable in many settings [34], [35]. We next show how *Session Rewrite* can automatically and fluidly reassign

[4]Protocols that eschew endpoint state for packet state are often termed stateless even though state exists in the packets.

```
1 % Sink: Send request message to Server.
2 message(@Client, Client, Server, Request) :-
3   interest(@Client, Server, Request).
4
5 % Server: Upon request, transition session state
6 session(@Server, Client, NewData) :-
7   message(@Server, Client, Server, Request),
8   session(@Server, Client, Data),
9   transition(@Server, Data, Request, NewData).
10
11 % ... and respond to Client.
12 message(@Server, Server, Client, NewData) :-
13   message(@Server, Client, Server, Request),
14   session(@Server, Client, NewData).
15
16 % Client: Consume response.
17 consume(@Client, Data) :-
18   message(@Client, Server, Client, Data),
19   interest(@Client, Server, Data).
20
21 % Query: What is consumed?
22 consume(@Client, Data)?
23
24 % Message forwarding used by both Server and Client.
25 message(@Next, Sender, Receiver, Payload) :-
26   message(@Current, Sender, Receiver, Payload),
27   nexthop(@Current, Sender, Next).
```

Listing 4. Original `SessionProg`, client-server roundtrip with session state. *session* initialization rules not shown.



(a) Original stateful server (b) Session state at proxy

Fig. 4. `SessionProg` network derivation graphs. The "loop" in 4a is stretched across the network to *proxy* in 4b.

session state to endhosts, packets, or proxies by simply filling in entries in a $rendezvous$ relation.

Listing 4 shows a client-request/server-response sequence with server responses based on session state. The $Client$ packages $interest$ as request $message$ tuples and sends these requests toward the $Server$ (lines 2-3). Upon receipt, $Server$ modifies $Data$ in its local $session$ according to the request and EDB relation $transition$, a relation capturing the protocol state machine.[5] It then returns a response $message$ to the $Client$ (lines 6-14). It is possible for $Client$ to make followup requests by expressing more $interest$ tuples, with responses dependent upon the state of $session$. Figure 4a shows the abstract network derivation graph corresponding to Listing 4 ($transition$ at $Server$ is not shown). Note that the queried relation in this case is actually at the client; we are interested in the client's status after a roundtrip communication with the server. Before discussing Session Rewrite, we first define an extension to the class of LR programs.

Definition 4.1: Two IDB relations a and b are **linearly mutually recursive** (LMR) if in the rule-goal graph, there is exactly one distinct path from a to itself that visits b one time and a zero times.

Definition 4.2: A program is a **linearly recursive program with linear mutual recursion** (LR-LMR program) if it is a LR program except for some relations that are LMR.

LR-LMR programs exhibit "linearity" equivalent to linearly recursive programs. The rule-goal graph path from a (b) to b (a) is linear (without branches), just as is the rule-goal graph path from a to a for LR relation a in a LR program. Our primary interest in LR-LMR programs for session state is when LMR relations a and b both participate in their own LR rules, and one (say b) has LR rules for which the location specifier does not change. In such a scenario, a is analogous to messages, and b is analogous to session state. It is this pattern upon which Session Rewrite operates. For the example in Listing 4, $message$ and $session$ map to a and b respectively.

Suppose that the queried relation is c. The main idea of Session Rewrite is to use MiM Rewrite as a subprocedure, and it is shown in Figure 4b. We let $message$ act as the queried relation, and apply MiM Rewrite to $session$ as if it participated in answer rules for $message$. First, $session$ generates bindings at $Server$ which get pushed down into $message$'s recursion until some rendezvous r (the proxy). $message$'s recursion also arrives at r, and MiM Rewrite operates as before, returning $message\_ans$ to $Server$. When this occurs, answer rules may derive new $message$ tuples. Because $message$ and $session$ are LMR, this in turn may derive new $session$ tuples. The new $session$ tuples generate new bindings, and are resent from $Server$ back to r to seek additional joins with $message$. The net effect is that r acts as proxy for $Server$'s $session$.

Proxy selection is determined by filling in the $rendezvous$ relation. Moreover, deciding among stateless, stateful and state proxy protocol variants is as straightforward as setting $rendezvous$ to $Client$, $Server$ or intermediate locations. The fully rewritten program after applying path-restrictions is shown in Appendix A. We generalize our example via the following corollary to Theorem 3.3.

Corollary 4.3: Session Rewrite is query preserving and path-restricted for LR-LMR programs.

Proof Sketch: LMR relations are essentially LR relations with aliasing. Therefore, using MiM Rewrite as a subprocedure is query preserving for "queried relation" a by Theorem 3.3. This means that if either a or b participates in answer rules for actual queried relation c, c is unaffected. If they do not participate, then query preservation is trivially true.

There are two additional features of typical session state that we also consider. First, session state changes are often based on a combination of input messages and protocol state machines. The example above encodes the state machine as the $transition$ relation. If the new session state is highly dependent upon the data in the input message (*e.g.,* $Request$ is a very selective join key in lines 6-6), then generating all possible bindings may result in a superset of $message$ that is very large. To mitigate this issue, we can choose to let some join key variables remain "free", even though their values are known. Choosing to exclude some join key bindings does not effect Corollary 4.3 as long as Constraint 1 is still observed.

Second, under some circumstances, we have the opportunity to reduce messages by piggybacking. For example, consider a scenario in which $proxy$ also was on the path from $server$ to $client$, in addition to being on the path from $client$ to $server$,

[5]Modification occurs via insertion of tuples with existing primary keys [13].

as in Figure 5. Then, rather than sending $message*$ backward from $server$ to $proxy$, we could piggyback $message*$ onto $message$ traveling from $server$ to $client$, and drop off the $message*$ portion at $proxy$. This is possible provided the following constraint.

Constraint 4 (Proxy Revisited): The $proxy$ must be visited on both the inbound path to and outbound path from the $server$.

This implies that any subsequent $message$ that goes from $client$ to $server$ will see the latest $session$ at $proxy$, thereby preserving Corollary 4.3.

B. Routing Proxies

Just as servers can become overloaded with too much session state, routers can likewise exceed their capacity for holding routing state. One solution is to let packets and proxies carry the routing state instead [21]. Our final rewrite, *Routing Rewrite*, exposes these options: it can reassign routing state to packets, proxies or some mixture of the two. Once again, this is done by filling in the $rendezvous$ relation.

Specifically, we apply Routing Rewrite to distance vector routing (DVR) and source routing (SR) which differ mainly in whether routing state resides in routers or packets. The prototypical message routing rule we have encountered thus far is line 7 of `BasicProg` (Listing 1). This resembles DVR, in that nodes send $message$ tuples to seek joins with $nexthop$. Conversely, SR sends $nexthop$ tuples to seek joins with $message$. Routing Rewrite transforms a DVR-style program to SR, or some hybrid of DVR and SR.

Like Session Rewrite, Routing Rewrite also uses MiM Rewrite as a subprocedure, acting as if LR relation a were the queried relation. Unlike Session Rewrite, it acts as if a's LR rules (such as $\mathcal{R}_1$) were answer rules. This means the base relations generate initial bindings, and the rest proceeds as described for MiM Rewrite. For Listing 1 where $message$ is the LR relation, this means $nexthop$ generates bindings and sends these backward according to the $nexthop$ relation.

With the base relations generating bindings, we can think of the base relation (which defines the network) as traversing itself. This effectively mirrors what happens in networking when data about the network (such as local connectivity information) is sent on the network. The following result follows the same proof by induction pattern as Theorem 3.3.

Corollary 4.4: Routing Rewrite is query preserving and path-restricted for LR programs.

As with the previous rewrites, *Decision Making* can select among alternatives simply by filling in the $rendezvous$ table after Routing Rewrite has been applied to the source program. To keep DVR, we set $rendezvous$ to the original sink. To convert to SR, we set $rendezvous$ to the source. To have some mixture of DVR and SR, we set $rendezvous$ to an intermediate location. The final result of Routing Rewrite on `BasicProg`, including path restrictions, is shown in Appendix A.

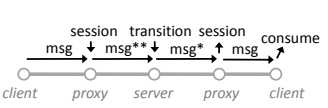


Fig. 5. Session Rewrite with piggybacking of $session$ on $message$.

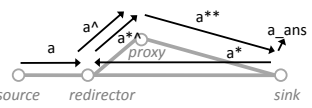


Fig. 6. Redirector points to off-path proxy.

C. Generalized Redirection

Thus far, our rewrites have relied solely upon on-path rendezvous and proxies. We have also extended MiM Rewrite and Session Rewrite to support redirection, a frequently used networking tool [36]. Redirection opens up possibilities for alternate paths.

With the generalized redirection modification, MiM Rewrite and Session Rewrite are able to expose every network location as a potential rendezvous and proxy candidate. The modification introduces an additional three rules for each recursive relation, and a new $redirect(\overline{redirector}, \overline{rendezvous})$ relation in the EDB. Its two attribute lists represent the (on-path) redirector, and the (off-path) rendezvous. The idea is that when any message headers of LR relation a or inverted relation $a*$ encounter a matching message header $\overline{redirector}$, the message body is combined with a new message header $\overline{rendezvous}$. This leads to a and $a*$ both being redirected to the off-path proxy, as in Figure 6. The relations $redirect$ and $rendezvous$ are now both available for the optimizer to populate. Off-path proxies require additional path-restrictions for correctness.

Constraint 5 (Off-path Proxy): A path must exist from the redirector to proxy and from proxy to sink.

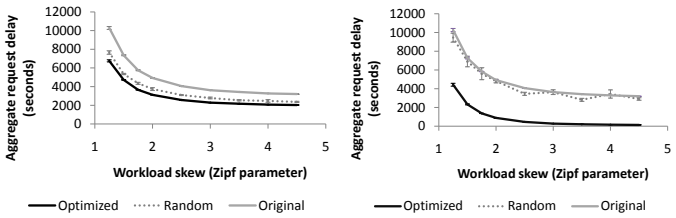
As with many of the other constraints, checks on Constraint 5 require data and rule-dependent analysis. *Decision Making* choose redirectors and proxies that observe this constraint.

V. Decision Making

The preceding section covered the application of three rewrites to `netlog` programs to expand their possible rendezvous and proxy choices. We now turn to *Decision Making*: the process of searching for the optimal strategy.

Inputs of *Decision Making* are network link costs and traffic profiles. Both the networking and database communities have extensively studied the problem of gathering such network workload and resource information [3], [15], [42]. In the context of `netlog`, input data are all represented as (synopses over) relations. This information can be monitored regularly, and if sufficiently different, can trigger re-optimization.

Outputs of *Decision Making* are tuples for the relations $rendezvous$ and $redirect$ initialized by *Rewriting*. We implemented exhaustive search algorithms for each rewrite. For MiM Rewrite, we also adapted a greedy heuristic from the CDN literature [32]. In principle, our rewrite-specific optimizations are replaceable by a general purpose dynamic programming optimizer, akin to those used widely by databases [33].



(a) Concentrated storage (b) Dispersed storage

Fig. 7. CDN rendezvous selection strategy performance under varying storage distributions and workloads.

A benefit of the `netopt` architecture is that the analysis and rewrite to identify the optimization opportunity is distinct from the mechanics of optimization. We have not focused on designing a better search algorithm for any specific scenario. Rather, we adopt an extensible framework which allows for the automated application of specific algorithms as appropriate [13], [28], [18]. This permits users to drop-in custom optimizers that best suite the task at hand.

VI. Prototype Evaluation

We built a prototype `netopt` system that performs *Analysis*, *Rewriting* and *Decision Making*. The implementation uses Evita Raced, an extensible database optimizer [13], and the resulting programs run on declarative networking platforms P2 and DSN [25], [11]. Our prototype still requires some user-assistance to link together the three steps.

We evaluate the `netopt` prototype in four scenarios involving the two settings of CDNs and WSNs discussed in Section I-A. We test against Emulab and Motelab testbeds [14], [30] for the CDN and WSN settings respectively, as well as in simulation. The objective of our experiments is to measure the change in application performance over original, unoptimized programs that do not adapt to workload and resource changes. The metric to quantify performance depends upon the setting. For CDNs, we consider content access delay while for WSNs, we consider energy usage.

In both scenarios, we see optimized programs outperforming unoptimized original programs. In the CDN setting, delay is decreased by as much as three orders of magnitude. In the WSN setting, radio operations which dominate energy consumption, are decreased by as much as one order of magnitude. In both settings, `netopt` effectively identifies and executes better strategies.

As with any optimizer study, the main point of our experiments is not to "invent" novel query plans (or in our case, protocol variants) that outperform well-known implementations from the literature. Rather, we wish to demonstrate that an optimizer can automatically choose variants that are well-suited to current input parameters, providing significant wins over well-known protocol variants that are not well-suited to the parameters.

A. CDN rendezvous selection

The goal of the CDN is to decrease access time of client requests while working within storage and topology constraints. Two popular CDNs, Akamai and Limelight, differ significantly in their storage layout [4], [23]. Akamai distributes content to tens of thousands of servers around the Internet, whereas Limelight maintains a few concentrated datacenters. We sought to model both during testing.

We tested by simulation and on Emulab. For simulation, we randomly generated a 200 node Internet Autonomous System (AS) topology with BRITE [27]. Some nodes are selected as content producers and others as content consumers. Producers and consumers are placed at nodes of low edge degree.

Content consisted of 150 unique items. Each consumer expressed a weighted demand for each content item. This query workload was modeled as a Zipfian distribution [17]. To experiment against varying workloads, we varied the skew of the Zipfian distribution.

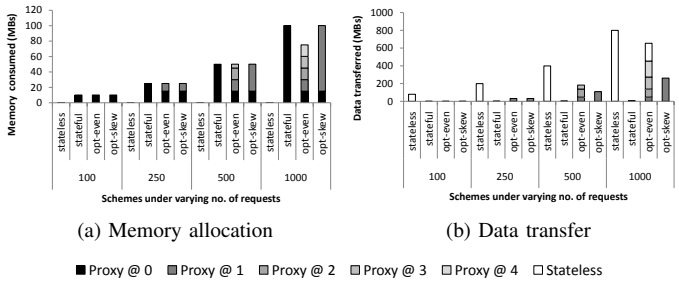
Each node was also assigned an amount of available storage. To experiment against varying resources, we used two schemes for storage assignment, mimicking Akamai and Limelight configurations. In the first Akamai-like scheme, available storage was spread evenly among nodes. In the second Limelight-like scheme, available storage was highly concentrated at a few nodes in the network. Well-connected nodes were favored to receive available storage. The amount of aggregate storage was the same in both configurations.

We experimented with four CDN assignment schemes. The first, the Original scheme, consists of `BasicProg` in Listing 1 in which all consumer requests go directly to the content producers; available CDN storage is not utilized.

The remaining three schemes all use the rewritten `BasicProg` produced by MiM Rewrite in Listing 3. They differed in the *Decision Making* scheme employed, and the extent to which the schemes use workload and resource information in planning the CDN. From the standpoint of the rewritten `BasicProg`, each scheme fed in its own $rendezvous$ and $redirection$ relation.

The second, Random scheme, consisted of randomly assigning content items to available storage. Here, resources are fully utilized, but the workload is not considered during assignment. The third, Optimized scheme, consisted of assigning content items to available storage such that consumer requests are serviced with lowest cost. The scheme used a greedy heuristic (by order of demand weight) for this assignment adapted from the literature since the optimal assignment is known to be in NP-Complete [32]. The fourth, the Exhaustive scheme, implemented the exponential version of the assignment algorithm. While the running time of Exhaustive was prohibitive on our test networks, we found that in the small settings, Exhaustive made assignments that were 8-12% better than those of Optimized.

Figure 7 shows the results of Original, Random and Optimized schemes under varying workloads and resources. Under the Concentrated storage configuration in Figure 7a, Random and Optimized performed 1.3-1.4× and 1.5-1.6×



(a) Memory allocation (b) Data transfer

Fig. 8. Server session state allocation strategy performance.

better than Original respectively as the workload varies from slightly skewed to highly skewed. Under the Dispersed storage configuration in Figure 7b, Random and Optimized perform 0.95-1.2× and 2.3-24.4× better than Original respectively. Optimized is able to outperform Original considerably under Dispersed because it is able to assign content items to edge storage sites where there is also heavy consumer demand. Random can even underperform Original with Dispersed since poor selections can be worse than doing nothing. On Concentrated, Optimized and Random start to converge since there are relatively fewer places to choose from. In all cases, increased skew leads to lower aggregate delay because there are fewer requests that access poorly placed content in highly skewed distributions.

We also ran the same experiments on modest ten node Emulab networks generated by BRITE. Random and Optimized outperformed Original by 1.5-1.9× and 2.8-3.3× with Concentrated, and by 1.2-2.3× 6.4-480× with Dispersed. The trends remained the same so the graphs are omitted.

These results indicate that MiM Rewrite can automatically find lower cost rendezvous points given original source program, consumer workload and network resources.

B. Server session state proxy selection

We next test the Session Rewrite. We use a five node linear Emulab network with node four making requests to node zero via nodes three, two and one. The workload is varied from 100 to 1000 requests, with each request requiring 1Kb of session state. Two storage configurations are used, Even and Skew. In Even, each node is allotted session storage of 15Mb, which is meant to represent prime main memory. In Skew, Node One is allotted 100MB for session state, whereas the other nodes are allotted 15MB. The Skew configuration models a scenario in which a resource rich proxy is located close to the server.

The optimization objective is to minimize the total data transfer while serving all requests. Three schemes are compared. In the first, Stateful, all session state is allocated at Node 0, regardless of whether the node storage constraint is surpassed. This corresponds to our original `SessionProg` of Listing 4. In the second scheme, Stateless, all session state is packaged in request and response messages. A minimal amount of storage is allocated at Node 0 to service these stateless requests. In the third scheme, Optimized, session state is assigned to proxies so as to minimize the total data transfer. This scheme tends to use as much storage available at proxies closer to the server, Node 0, before using storage further from the server. The Optimized scheme runs the rewritten `SessionProg` shown in Listing 4.

Figure 8 shows the memory allocation and data transfer of each scheme under varying numbers of requests and storage configurations. As expected, Stateless maintains an almost negligible amount of storage usage across all nodes regardless of the number of requests, while its amount of data transfer grows very rapidly since it must package all of its request state in packets. Conversely, as seen in Figure 8a, storage usage under Stateful at Node 0 scales with the number of requests, well surpassing the 15MB constraint under 250 or more requests. On the other hand, the amount of data transfer with Stateful remains low even when there are many requests. Neither Stateful nor Stateless take advantage of the potential to use other nodes as proxies in the network, and therefore do not act differently when storage configurations change.

The Optimized scheme is able to take into account varying storage configurations. In Figure 8a, the "opt-even" and "opt-skew" labels show the resulting memory allocation on each node when the Session Rewrite optimizes against Even and Skew configurations respectively. At 100 requests when the storage limit is not yet reached, Optimized behaves just like Stateful. At higher request counts, the constraint is respected by the Optimized scheme, and storage is allocated from neighboring proxies rather than at node zero. Each segment of the stacked bars in Figure 8b indicates the amount of data transfer as a result of session state held at the corresponding proxy. For a given request workload, the optimized version transfers less data than Stateless, but more data than Stateful (while respecting storage constraints). This hybrid of stateless and stateful is a compromise when storage constraints are present. Furthermore, the optimizer is able take advantage of the resource-rich proxy in the Skew configuration, and transfer lower amounts of data by using Node One more when storage limits become an issue at higher request counts.

The use of proxies does come at a cost: there is a small penalty of two bytes per session that is incurred for both storage and in each packet transfer. These two bytes are needed for the join parameters which relink a request with its session state at the proxy.

The "Stateless" and "Proxy @ 4" legend labels in Figure 8b both indicate session state stored at Node Four. The "Proxy @4" label indicates session state that is stored as part of the 15MB allotted storage visible to the optimizer, whereas the "Stateless" label applies to session state stored in storage separate from the 15MB allotment invisible to the optimizer *e.g.,* in a separate user processes. We include this comparison because it is not uncommon for "stateless services" to delegate state management to other processes [34], [1] Optimized Even engages this option when there are 1000 requests in order to avoid exceeding storage constraints.

The results indicate that the rewritten `SessionProg` can have its server session state automatically assigned to proxies

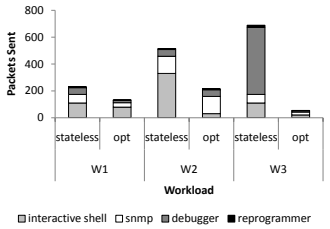


Fig. 9. Packets sent by sensornet session state strategies.

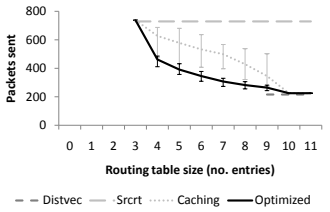


Fig. 10. Sensornet routing state placement strategy performance.

and packets effectively by an optimizer, lowering overall data transfer versus stateless variants. At the same time, the optimizer can automatically respect storage constraints, unlike the original stateful `SessionProg`.

C. Sensornet session state proxy selection

Next, we measure the effectiveness of Session Rewrite on WSN programs. The optimizer attempts to minimize packets sent and received, since radio operations are often the most power-intensive activity on sensornets.

Four traditionally stateful services that have been implemented on sensornets were chosen from the literature: a network Reprogrammer, a network Debugger, an SNMP-like service, and an Interactive Shell service [20], [43], [38], [9]. For each service, we estimated the state required as the service's RAM footprint as reported in the literature. These were 0.15Kb, 1Kb, 1.2Kb and 2.2Kb for Reprogrammer, Debugger, SNMP, and Interactive Shell respectively. For testing purposes, we ran placeholder programs that demanded the same amount of session state as the original services.

These services are generally auxiliary to the main sensornet application. Therefore, the typical usage model is that it is highly desirable, though not critical, to deploy these services alongside the main application. We worked with two scenarios: the first in which the main application consumed 8Kb, and the second in which the shell consumed 5Kb, both of which we estimated from prior experience with sensornet applications. Given the mote platform we were using, this left 2Kb and 5Kb of main memory for our desired services [31].

We deployed Stateful, Stateless and Optimized programs on the Motelab testbed. The Stateful program consisted of the session state of all four services plus the main application. The Stateless program consisted of the main application, but no session state. Rather, state is transported in packets, whose data payload is a typical 20Kb in size [1]. The Optimized program consisted of the main application plus a portion of each service's session state as allocated by the optimizer, with the rest pushed into packets. In each case, requests are made from a base station node across five hops to a node in the testbed that runs either Stateful, Stateless or Optimized.

The workload consisted of varying the distribution of calls made to each of the four services. We considered three synthetic workloads: W1, an evenly distributed workload; W2, a network monitoring workload in which SNMP and Interactive Shell were called two and three times more; and W3, a debugging workload in which Debugger and Reprogrammer were called two and ten times more.

The packets sent for the 2Kb storage limit scenario are shown in Figures 9. The number of packets sent for Optimal are 1.7-12.6× lower than that for Stateless, with the difference increasing as the workload becomes more skewed in W2 and W3 (Figure 9). Optimized allocates the most frequently called services' session state to keep on the node, thus lowering the amount of packet state necessary. Stateless, on the other hand, uses very little of the available memory, and hence is required to send more packets. It is not possible to test the packets sent for Stateful since the session state exceeds availability.

At the 5Kb storage limit, each of the four services has enough memory for its entire session state. Therefore, Stateful and Optimized perform similarly in packets sent and storage allocated. Stateless, which does not change operationally with an increased storage limit, sends the same high number of packets as before. The graph is omitted for lack of space.

D. Sensornet routing state placement

Lastly, we look at routing state placement in the sensornet, and measure the ability of Routing Rewrite and optimization to choose routing state proxies. We chose a Motelab network of four hops starting from the base station. Storage is constrained such that nodes only have space for a limited number of routing entries, varying from three to eleven. Typical sensornet routing services contain four entries [1]. The base station node initiated sends to nine destinations located four hops away in the network according to a Zipfian distribution.

Figures 10 demonstrate the results of Source Routing (SR), Distance Vector Routing (DVR), Caching, and Optimized. SR is essentially stateless, and is able to route with very few available routing entries, albeit at more packets sent. On the other hand, DVR only routes when it has enough space for all nine destinations (such that semantics were equal). Caching uses the hybrid approach of SR as the default case and residual space for DVR routing entries as requests arrive. It tends to have very high variance in packets sent due to variability of which request arrives first for caching. This variability would decrease were Caching to allow for cache eviction. Optimized considers the workload such that the hotter destinations receive higher priority as DVR entries. As a result, it tends to achieve the lowest number of packets sent at all routing table sizes.

We have modeled each source route segment to correspond to one packet when in SR mode. This is a conservative assumption, and partially based on the inability of Session Rewrite and the DSN runtime to bundle multiple tuples into a single packet. Were segments to be bundled, the difference between packets sent for SR and DVR would decrease proportionally to the number of segments fitting in one packet.

VII. Discussion

We chose to focus on rendezvous and proxy placement for a number of reasons: (a) they are fundamental to multiple "layers" of both networks and distributed systems, (b) decisions on these two fronts form key differentiators between many

implementation alternatives in networks, and (c) declarative networking bring these issues into sharper focus than they had been in other programming models. In this section we mention several other scenarios to which our results are readily applicable, and ways in which `netopt` can interoperate with legacy network implementations.

A. Other Application Scenarios

Packet filtering is used to eliminate unwanted traffic based on rules about addressing, content, and volume. If a recipient node x wishes to ensure that packets are filtered on its behalf, it can either (a) receive all packets addressed to it, and filter them before processing them further (filter at receiver), (b) force all senders to evaluate packet filters (filter at sender), or (c) appoint a proxy or proxies in the network to intercept traffic between and senders and the recipient (filter at proxy). The choices amount to selecting a node or nodes where filtering rules and the messages destined for x will rendezvous; these nodes must maintain a copy of the packet filer rules for x (and must be trusted by x by some means, typically cryptographically).

Work on providing scalable Internet quality of service has also investigated using stateless protocols and proxies in place of stateful ones [35]. In these scenarios, a primary goal is to permit the large majority of core Internet routers to remain stateless while pushing quality of service state into packets and proxies at edge routers. These demonstrated that a variety of objectives such as per flow bandwidth fairness, admission control, and route pinning could all be achieved while shifting router state to packet state.

B. Interoperability

Our focus is on cooperative scenarios where individual nodes are non-adversarial. Often, this corresponds to situations in which the nodes of the network fall under a single administrative domain, as is typically the case for CDN and WSN deployments. In cooperative scenarios, individual nodes are not expected to deviate from global plans installed by a network optimizer. Therefore, interoperability may be an initial concern for `netopt`. We categorize interoperability in terms of packet, execution and state, and argue that the deductive database programming model is not a hindrance to interoperability in these areas. In fact, the model may even ease interoperability by natively supporting proxies, a common interop mechanism.

In terms of packet interoperability, database schema definitions can serve the role of packet format definitions. Schemas specify field layouts and data types for relations, and since packets are tuples in the `netlog` programming model, the application is straightforward.

In terms of execution interoperability, it is true that a `netlog` program and traditional program that do not understand each other's protocols will not be able to interoperate. However, this is no different than the current incompatibility between TCP and UDP, or between any other two sets of protocols not designed with the same properties in mind.

A legitimate concern is state interoperability. That is, the optimization process changes which nodes carry what state such that an outside party wishing to interoperate may receive incomplete or unexpected protocol state. This is an issue that manually optimized variants such as Trickles TCP face when interfacing with originals such as regular TCP. On this point, the natural solution is to bridge the two incompatible protocols via proxy. Here, `netopt` can start to offer some guidance.

The `netopt` programming model is capable of exposing programming constraints to the optimizer. With additional programmer hints, the optimizer can force certain relations to remain stateful. This general idea of not only optimally placing proxies, but also of forcing stateful proxies at certain points in the network is already native to `netopt`. This starts to offer a path for incremental deployment, very much following the spirit of manually engineered solutions to interoperability: use of proxies [40].

VIII. Related Work

Related work stems from both networking and databases.

A. Network Protocol Optimization

Prior work in network protocol optimization generally focuses on packet processing performance on the single node, usually by adapting techniques from general compiler optimization [10], [6], [19], [29], [22]. On the other hand, our focus is automated multi-node protocol optimization.

Protocol compilers take high-level protocol specifications and generate protocol implementations. Often, there is a need to optimize protocol implementations built in this manner to achieve comparable performance to manually engineered solutions. Two optimizers in this vein are HIPPCO for the Esterel language [10] and the Promel++ optimizer for the Promela++ language [6]. Both employ the use of techniques to increase single-node packet processing performance: inlining, outlining, code cloning, rearranging branches to increase cache hit rates, and IPC to function call conversion. These same traditional compiler techniques are also readily applicable to protocols programmed in systems languages like C and C++ [29], [22].

While the prior work on protocol optimization tackles a different optimization problem than the ones addressed in this work, they also suggest the need for workload knowledge to accurately optimize for the common case [29], [10], [6]. In this earlier work, protocol writers annotated their code with workload-specific knowledge (such as branch predictions), whereas in this work, workload knowledge is based upon database-driven statistics gathering.

Several efforts have attempted to enable greater network flexibility. Active networks research moved aggressively to introduce greater programmability into networks [41]. Our work introduces a limited amount of network reprogramming, driven by optimizer decisions rather than node-level code injections. Like our work, i3 identifies rendezvous and proxy selection as fundamental to network design, and provides great flexibility for their selection [36]. Unlike our work, i3 does not aim to optimize their selection from program source.

B. Database Query Optimization

Several of the specific network optimization mechanisms introduced in this work can be viewed as generalizations of query processing and optimization mechanisms familiar to the database community. Changing rendezvous is conceptually very similar to reordering database join operations. Therefore, as we explore rendezvous, join reordering underlies many of the mechanisms employed in this work. System R popularized the ideas of optimizing join ordering with respect to disk IO, CPU, and table statistics [33]. The current work fundamentally adopts the same optimization framework, extended to the networked setting. This is not the first attempt to do so [37]. We significantly broaden the scope of what can be reordered, and thus what reordering is capable of. Specifically, this is enabled by treating the traditionally distinct "application data" and "networking data" all in the same relational context.

In the past, deductive database query optimization focused on combining "push" with "pull" query processing [39]. The main result here is the Magic Sets algorithm that transforms programs to take advantage of the benefits of pull processing while executing in a push context [7]. The work of [26], [24] extended this to the networked setting, specifically applying an entirely pull processing approach to the example of routing as in Section IV-B. In contrast, this work suggests that hybrids between top-down and bottom-up processing offer the best cost for many practical networking scenarios.

We suspect it is possible to generalize MiM Rewrite to all recursion, just as algorithms for LR have been subsumed by the Magic Sets algorithm [39]. However, our experience indicates that LR is the most common, especially in networking. This also echoes the remarks of [39] for traditional Datalog.

Our work builds upon the results of the P2 project [26], [25], [24] from which we adopt the declarative networking approach in whole. Our contribution lies in the resource and workload-aware network optimizations.

IX. Conclusion

As network workloads and resources continue to diversify, one-size-fits-all network protocols are increasingly infeasible, while custom solutions require careful crafting for each environment. We investigated automatic program analysis, rewriting and optimization of network protocols along dimensions of rendezvous and proxy selection. Our study indicates that under a variety of settings, an informed optimizer can choose program executions that are much better than that of the original source program.

References

[1] Tinyos. http://www.tinyos.net.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Computer Science Press, 1995.

[3] C. Aggarwal and P. Yu. A survey of synopsis construction in data streams. *Data Streams: Models and Algorithms*, pages 169–208, 2006.

[4] Akamai. http://www.akamai.com.

[5] T. Anderson, D. Blumenthal, D. Casey, D. Clark, D. Estrin, L. Peterson, D. Raychaudhuri, J. Rexford, S. Shenker, and J. Wroclawski. Geni: Conceptual design project execution plan. In *GENI Design Document GDD-06-07, January 2006. http://www.geni.net/GDD/GDD-06-07.pdf.*

[6] A. Basu, J. G. Morrisett, and T. von Eicken. Promela++: A language for constructing correct and efficient protocols. In *INFOCOM*, 1998.

[7] C. Beeri and R. Ramakrishnan. On the power of magic. In *PODS*, 1987.

[8] D. Bernstein. Syn cookies. http://cr.yp.to/syncookies.html.

[9] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He. An interactive unix shell for low-end sensor nodes with liteos (demo). In *SENSYS*, 2007.

[10] C. Castelluccia, W. Dabbous, and S. O'Malley. Generating efficient protocol code from an abstract specification. *IEEE/ACM Trans. Netw.*, 5(4):514–524, 1997.

[11] D. C. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *SENSYS*, Nov 2007.

[12] F. A. Chudak and D. P. Williamson. Improved approximation algorithms for capacitated facility location problems. *Lec. Notes in Comp. Sci.*, 1610, 1999.

[13] T. Condie, D. Chu, J. Hellerstein, and P. Maniatis. Evita raced metacompilation for declarative networks. In *VLDB*, 2008.

[14] E. Eide, L. Stoller, and J. Lepreau. An experimentation workbench for replayable networking research. In *NSDI*, 2007.

[15] C. Estan, S. Savage, and G. Varghese. Automatically inferring patterns of resource consumption in network traffic. In *SIGCOMM*, 2003.

[16] M. Franklin and S. Zdonik. Data in your face. In *SIGMOD*, 1998.

[17] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with coral. In *NSDI*, 2004.

[18] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, 1993.

[19] D. Hernek and D. P. Anderson. Efficient automated protocol implementation using rtag. Technical Report UCB/CSD-89-526, EECS Department, University of California, Berkeley, Aug 1989.

[20] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SENSYS*, 2004.

[21] D. B. Johnson and D. A. Maltz. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*, volume 353. 1996.

[22] E. Kohler, R. Morris, and B. Chen. Programming language optimizations for modular router configurations. In *ASPLOS-X*, 2002.

[23] Limelight. http://www.limelightnetworks.com.

[24] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking with distributed recursive query processing. In *SIGMOD*, 2006.

[25] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *SOSP*, 2005.

[26] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: Extensible routing with declarative queries. In *SIGCOMM*, 2005.

[27] A. Medina, A. Lakhina, I. Matta, and J. Byers. Brite: An approach to universal topology generation. In *MASCOTS*, 2001.

[28] G. Mitchell, U. Dayal, and S. B. Zdonik. Control of an extensible query optimizer: A planning-based approach. In *VLDB*, 1993.

[29] D. Mosberger, L. L. Peterson, P. G. Bridges, and S. O'Malley. Analysis of techniques to improve protocol processing latency. In *SIGCOMM*, 1996.

[30] Motelab. http://motelab.eecs.harvard.edu.

[31] J. Polastre, R. Szewczyk, and D. Culler. Telos: enabling ultra-low power wireless research. In *IPSN*, 2005.

[32] L. Qiu, V. N. Padmanabhan, and G. M. Voelker. On the placement of web server replicas. In *INFOCOM*, 2001.

[33] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[34] A. Shieh, A. Myers, and E. G. Sirer. Trickles: A stateless network stack for improved scalability, resilience and flexibility. In *NSDI*, 2005.

[35] I. Stoica. *Stateless Core: A Scalable Approach for Quality of Service in the Internet*. PhD thesis, Carnegie Mellon University, 2000.

[36] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. *IEEE/ACM Trans. Netw.*, 2004.

[37] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa. *VLDB J.*, 1996.

[38] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. 2005.

[39] J. D. Ullman. *Principles of Database and Knowledge-Base Systems. Volume 2, The New Technologies*. Computer Science Press, 1989.

[40] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes No Longer Considered Harmful. In *OSDI*, 2004.
[41] D. Wetherall. Active network vision and reality. In *SOSP*, 1999.
[42] K. Xu, Z.-L. Zhang, and S. Bhattacharyya. Profiling internet backbone traffic. *SIGCOMM CCR*, 35(4):169–180, 2005.
[43] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: a comprehensive source-level wsn debugger. In *SENSYS*, 2007.

APPENDIX

```
1 % Sink: Send request to Proxy.
2 request(@Sink, Sink, Src) :- interest(@Sink, Src).
3 request(@Next, Sink, Src) :- request(@Curr, Sink, Src),
    nexthop(@Curr, Src, Next), -rendezvous(@Curr, Sink, Src).

5 % Source: Send session to Proxy
6 request*(@Src, Src, Sink, Data) :- session(@Src, Sink, Data),
7 request*(@Prev, Src, Sink, Data) :-
    session(@Curr, Src, Sink, Data), nexthop(@Prev, Sink, Curr),
    -rendezvous(@Curr, Sink, Src).

9 % Proxy: Combine request and session and send to Source
10 request**(@Curr, Sink, Src, Data) :- request(@Curr, Sink, Src),
    request*(@Curr, Src, Sink, Data),
    rendezvous(@Curr, Sink, Src).
11 request**(@Next, Sink, Src, Data) :-
    request**(@Curr, Sink, Src, Data), nexthop(@Curr, Src, Next).

13 % Source: Upon request, transition session state
14 session(@Src, Sink, NewData) :-
    request**(@Src, Sink, Src, Data),
    transition(@Src, Data, NewData).
15
16 % ... and respond to Sink.
17 message(@Src, Src, Sink, Data) :-
    request**(@Src, Sink, Src, Data).
18 message(@Next, Src, Sink, Data) :-
    message(@Curr, Src, Sink, Data), nexthop(@Curr, Sink, Next).
19
20 % Sink: Consume response.
21 consume(@Sink, Data) :- message(@Sink, Src, Sink, Data),
    interest(@Sink, Src).
22
23 % Query: What is consumed?
24 consume(@Sink, Data)?
```

Listing 5. Rewritten `SessionProg`, a client-server roundtrip with session state proxy.

```
1 % regular message routing until rendezvous
2 message(@Next, Src, Sink, Data) :-
    message(@Curr, Src, Sink, Data), nexthop(@Curr, Sink, Next),
    Curr != Sink, -rendezvous(Next).
3
4 % building source route: self-traversal backward
5 nexthop*(@Curr, Curr, Sink, Next) :- rendezvousIn(@Curr),
    -rendezvousIn(@Next), nexthop(@Curr, Sink, Next).
6 nexthop*(@Curr, Curr, Sink, Next) :- rendezvousIn(@Curr),
    rendezvousIn(@Next), nexthop(@Curr, Sink, Next),
    nexthop*(Curr, Next, Sink, _).
7 nexthop*(@Prev, Link1, Sink, Link2) :-
    nexthop(@Prev, Sink, Curr),
    nexthop*(@Curr, Link1, Sink, Link2).
8
9 % using source route: self-traversal forward
10 nexthop**(@Next, Link1, Sink, Link2) :-
    nexthop*(@Curr, Link1, Sink, Link2), Curr != Sink,
    message(@Curr, Src, Sink, Data), nexthop(@Curr, Sink, Next).
11 nexthop**(Next, Link1, Sink, Link2) :-
    nexthop**(Curr, Curr, Dest, Next),
    message(Curr, Src, Dest, Data),
    nexthop**(Curr, Link1, Sink, Link2), Link1 != Curr.
12 message(Next, Src, Sink, Data) :-
    nexthop**(Curr, Curr, Dest, Next),
    message(Curr, Src, Dest, Data),
    nexthop**(Curr, Link1, Sink, Link2), Link1 != Curr,
    -nexthop(Curr, Link1, Sink, Link2).
```

Listing 6. Rewritten Routing Rewrite, a DVR-SR hybrid.