Automating Rendezvous and Proxy Selection in Sensornets

David Chu
EECS Computer Science Division
University of California, Berkeley, CA 94720
davidchu@cs.berkeley.edu

Joseph M. Hellerstein
EECS Computer Science Division
University of California, Berkeley, CA 94720
hellerstein@cs.berkeley.edu

ABSTRACT

As the diversity of sensornet use cases increases, the combinations of environments and applications that will coexist will make custom engineering increasingly impractical. We investigate an approach that focuses on replacing custom engineering with automated optimization of declarative protocol specifications. Specifically, we automate network rendezvous and proxy selection from program source. These optimizations perform program transformations that are grounded in recursive query optimization, an area of database theory. Our prototype system implementation can automatically choose program executions that are as much as three, and usually one order of magnitude better than original source programs.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design; H.2.4 [**Database management**]: Systems

General Terms

Algorithms, Design, Performance

Keywords

Sensor Networks, Network Optimization

1. INTRODUCTION

Networks are growing increasingly diverse – or equivalently, diverse networks are increasingly inter-networked. The causes of this diversity stem from both novel workload demands from above and new resource availability from below. Wireless sensornets exemplify the situation. From above, we are encountering new applications that exhibit audio processing, video-on-demand and distributed feedback-based control. From below, we are composing infrastructure from satellites, cellular networks, urban WiFi, and short-range radio like 802.15.4 and bluetooth.

Sensornet 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 is poised to outstrip the ability to address each combination individually.

As an alternative approach, we show that sensornet programs written as *declarative specifications* can be optimized by a database-inspired optimizer. Earlier work showed declarative specifications can compactly express many sensornet problems, from networking services such as localization, routing and data dissemination to end-user queries. Furthermore, these specification can be compiled into operational systems that achieve comparable performance to traditional, imperatively programmed implementations [10].

In this work, we focus on networking optimizations that address prototypical networking rendezvous and proxy placement questions: Where should messages from communicating parties rendezvous – via push, pull or some of both? 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? Sensornet system builders often wrestle with these choices in concrete instances to achieve better performance. Section 1.1 discusses some of these situations in detail. Our optimizer, `netopt`, mitigates the need for case-by-case consideration of rendezvous and proxy selection. This is timely in the WSN environment, given increasingly diverse and varying workloads and resources.

In addition to the utility of our optimizations, a conceptual contribution of our work is in exposing the congruence between network design and recursive query optimization, a traditional topic in database theory. Specifically, optimal network rendezvous and proxy selection is roughly analogous to cost-based selection pushing in the presence of recursive queries.

To examine the utility of our network optimizations, we apply them to both simulation and testbed WSNs. In simulation, gains are by as much as three orders of magnitude. On testbed WSNs, gains are by as much as one order of magnitude. In both settings, `netopt` effectively identifies and executes better strategies.

Section 1.1 takes a closer look at our two chosen application scenarios. Section 2 introduces our distributed and recursive query language and offers initial attempts at network optimization. Section 3 discusses the main rendezvous optimization in detail. Section 4 extends this to proxy placement optimization. Section 5 discusses their execution on our implementation platforms. Section 6 reports on prototype implementation and deployment of our optimizations.

1.1 Rendezvous and Proxies in WSNs

One predominant application class for WSNs is event detection and distribution. In the naive variant, an event source sends event notifications to the sink *i.e.,* rendezvous only at the sink. Yet rendezvous at other locations in the network is conceptually possible and often beneficial. For example, if many events are generated but only a few are of interest to the sink, it may be more energy efficient to pass the sink's selection criteria (part of the way) to the source. Furthermore, in the case of multiple source and sink pairs, limited node buffer space may preclude every pair from using its preferred rendezvous. The optimization problem is akin to ones encountered outside WSNs *e.g.,* in PubSub [16] and Content Distribution Networks [26] where it is known to correspond to the NP-complete facilities location problem [11]. We show how `netopt` can automatically identify naive cases from program source, rewrite them to expose rendezvous flexibility, and assign lower cost rendezvous.

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? And who makes these decisions? The service designer implementing something like interactive login will not know the needs and constraints of a each specific deployment. On the other hand, the end system deployer can not be expected to be intimately familiar with reconfiguring the protocols of every packaged service. This leads to compromises, in which conservative service designers minimize node state at the expense of increasing in-flight packet state. Since the radio is frequently the most power-intensive hardware unit, the increased communications directly decrease the overall network lifetime. We tackle this problem with automated techniques for exposing and optimizing proxy placement.

```
% Prepare for transmission
message(@Source, Source, Sink, Data) :-
   produce(@Source, Data),
   nexthop(@Source, Sink, Next).

% Route message to next hop parent
message(@Next, Source, Sink, Data) :-
   message(@Current, Source, Sink, Data),
   nexthop(@Current, Sink, Next).

% Receive if message is of interest
consume(@Sink, Data) :-
   message(@Sink, Source, Sink, Data),
   interest(@Sink, Data).

% What is consumed?
consume(@Sink, Data)?
```

Listing 1: Original `BasicProg`, event distribution from source to sink with filtering by interest.

2. EXAMPLE PROGRAM OPTIMIZATION

Throughout this work, the deductive database programming model of Datalog [24] is used as a means to demonstrate the concept of automated analysis, rewriting and optimization. The specific dialect, `netlog`, is convenient due to its immediate display of recursion, which we heavy utilize. `netlog` is a subset of OverLog [19].[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.

2.1 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,

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

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.

To extend to distributed systems, every tuple is stored at the network node indicated by its first attribute, the location specifier (denoted with the "@" symbol). Each node holds a *partition* of each relation keyed on the first attribute. For instance, a node's partition of a relation like *nexthop* can reflect its local routing table.[2] When a rule involves tuples across partitions, communication between nodes occurs to access the necessary data. As we shall see, much of the work in this paper is targeted at automatically optimizing partition accesses.

In `BasicProg`, the *produce* relation contains pairs of *Source* and *Data* attributes. When these tuples are *joined* against *nexthop* tuples, initial *message* tuples bound for *Sink* are generated (lines 2-4). Joining produces an output tuple every time there exist tuples in the body that possess equal attribute values when the attribute names are the same. Attributes with matching names are *join keys*. For example, consider the first rule of `BasicProg`: a *message* tuple's first attribute takes on some value *Source* only when there exist tuples in *produce* and *nexthop* whose first attribute values are both *Source*.

The data is routed via the second rule of `BasicProg` by recursively defining the contents of the *message* relation with respect to 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 via the third rule of `BasicProg` (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).

2.2 Pushing Selections One-Hop

As an example, consider a two node network x and y represented by a previously-defined set of facts (sometimes called an "Extensional Database (EDB)") $\mathcal{D}$ consisting of three relations:

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

The EDB is the set of relations that are never in the head of any rule; its tuples are defined exogenously, perhaps via a data structure in a persistent store. Conversely, the Intensional Database (IDB) is made of the derived relations that occur in rule heads. The IDB of $\mathcal{D}$ is:

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

In `BasicProg`, *produce* and *interest* rendezvous at a node x via sending of a *message* from some node 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).
```

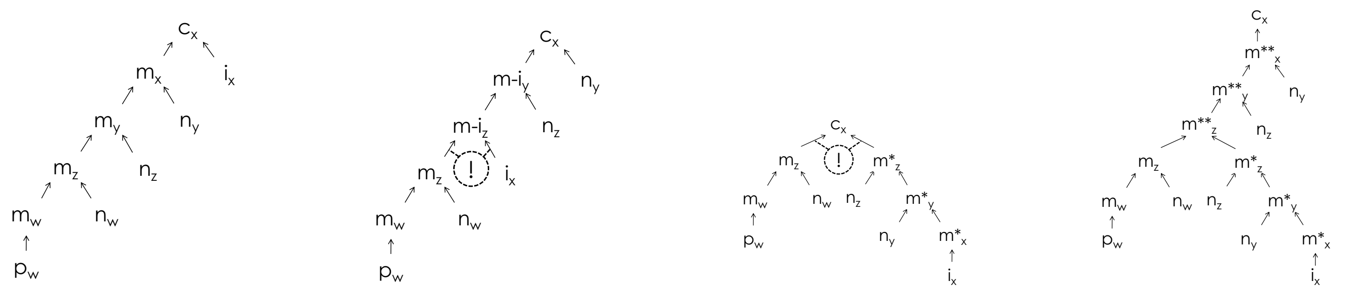
The first rule prepares *interest* tuples as *message*$*$ tuples. The second rule passes *message*$*$ *backward* along *nexthop*, and is similar to how *message* was routed in Listing 1. The third rule derives *consume*. For the one-hop network, these rules produce the desired result of rendezvous at y, with the queried *consume* at x. As a result, we have "pushed" the selection condition *i.e., interest* back to *produce*.

2.3 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, which is similar to a technique mentioned in [20]. 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

[2]Listing 1 assumes the tuples of *nexthop* are given to us; earlier work showed how additional rules can be used to define routing tables [10, 20].



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

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

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 2.2. $produce_w$ 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. To resolve this issue, we can "package up" $consume_x$ as a new relation *message∗∗* and send it along the network topology via a path we already know about from z to x. Figure 1d shows this as part of the fully path-restricted network derivation graph with rendezvous at z. This is just one possible rendezvous choice. The "Meet-in-the-Middle" *MiM Rewrite* we discuss next transforms input programs to expose many possible rendezvous choices.

3. MiM Rewrite

The `netopt` network optimization architecture executes in three stages:

- *Analysis* identifies optimization opportunities from input programs. We show how to identify rendezvous and proxy selection opportunities.
- *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.

This section first sets forth the correctness criteria of any `netopt` optimization, and describes the MiM Rewrite procedure precisely in terms of its analysis and rewrite phases. 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.*

Note that neither of these definitions constrains the contents of the IDB in general, only the queried relations.

3.1 Analysis

Analysis identifies certain rules and relations as rewrite components. We first introduce some terminology from classic work in the deductive database literature [31].

Definition 3.3. *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 3.4. *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 3.5. *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 3.6. *A relation a is an* ***LR relation*** *if a is the head of an LR rule. The other relations in the body of the LR rule are* ***base relations****.*

Without loss of generality, we can restrict our discussion to scenarios in which the LR rule body contains only one base relation.[3] Our focus on networking programs leads us to consider the following type of LR rule.

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

In the rule, the value b_i determines the new location specifier. Therefore, 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 b as defining a network for the LR relation a to "hop along". Both LR and base relation identification can be accomplished by traversing the rule-goal graph.

Looking at the attribute variables in the example, note that each d_i can correspond to any a_i or b_i to get data values from the input to the output. Furthermore, b_i's can correspond to a_i's to capture join conditions between a and b.

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

Definition 3.7. *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.

3.2 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.

[3]When this is not so, it is straightforward to rewrite the program to include a rule that derives a single base relation by joining multiple base relations.

A preliminary procedure of the MiM Algorithm, common in the deductive database literature [2], 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, a *binding list* for each recursive relation a is produced. A binding list α is a sequence of "b"s (bound) and "f"s (free), with each character representing an attribute of a. An attribute of a is bound ("b") if possible values are (1) already known since they are join keys with EDB relations, and (2) useful since the join happens in an answer rule. Informally, the binding list is a template that guides the search for derivations that might actually matter to the queried relation. 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.

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

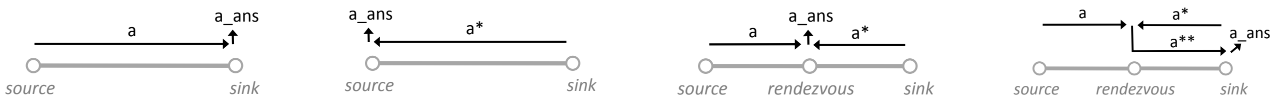
In this basic yet common case, the binding list α is assigned as follows: α_i is "b" if a_i joins with some e_j. Otherwise α_i is "f". Furthermore, with some trivial variable reordering, we can safely assume that α is a sequence of "b"s followed by a sequence of "f"s.

With these preliminaries, the 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, and consists of a letter and optionally a digit, *e.g.,* $\overline{a1}$. The letter indicates that the size of the list is the number of attributes of the corresponding relation *e.g.,* $\overline{a1}$ is a variable list of size Na. The digit is just an identifier.

To manipulate variable lists, we use three functions. *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".

Each variable list originates 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 $\overline{a1}$.

The 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.



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

Figure 2: Steps of MiM Algorithm

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$(\overline{b})$.

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$(\overline{b})$.
   $\mathcal{R}_4$ 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$(\overline{b})$, −r$(\overline{a2_b})$.
   $\mathcal{R}_{3.1}$ a∗$(\overline{a2_b}, \overline{a0_b})$:− a∗$(\overline{a1_b}, \overline{a0_b})$, b$(\overline{b})$, −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$(\overline{b})$.
   $\mathcal{R}_6$ a_ans$(\overline{a0_b}, \overline{a3_f})$:− a∗∗$(\overline{a0_b}, \overline{a0_b}, \overline{a3_f})$.

Listing 2: MiM Algorithm

The MiM Algorithm consists of three main steps traced by Figure 2. Figure 2a shows the input program as an *abstract 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 Magic Sets algorithm from database theory [5]: pushing selection past recursion. This is done by constructing $a*$ to recurse backward from sink to source (Figure 2b). In networking terms, pushing down selections in this setting can be thought of as a sink-initiated "pull" execution vs. the original source-initiated "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 in an EDB relation. These are simply copied over 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$(\overline{e})$. 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, used to remember the original join keys for 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 point 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 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. We next present an example application of MiM Algorithm.

3.3 Example Application of MiM Algorithm

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

```
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.

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 5. In the companion technical report [8], we establish the correctness of MiM Rewrite by proving the following theorem.

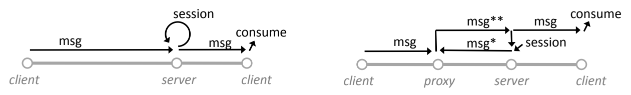
THEOREM 3.8. *The MiM Rewrite is query preserving and path-restricted.*

4. 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.

4.1 Session Proxies

Many protocols and services maintain per-conversation session state at endpoints. However, a server may get many simultaneous connections, or multiple services might need to coexist. Either case may exhaust session state buffer space. As a result, a systems builder 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 [27], and is often handled manually in WSNs. We next show how *Session Rewrite* can automatically and fluidly reassign session state to endhosts, packets, or proxies by simply filling in entries in a *rendezvous* relation.



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

Figure 3: Abstract network derivation graphs for session state placement alternatives. The "loop" in 3a is stretched across the network to *proxy* in 3b.

Figure 3a shows the abstract network derivation graph corresponding to a client-request/server-response sequence with server responses based on session state. At the server, both the current *message* and *session* help to derive the next logical *message* and *session*. Also, the queried relation in this case, *consume*, 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 before returning to a.*

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.*

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 Figure 3a, *message* and *session* map to a and b respectively.

The main idea of Session Rewrite is to use MiM Rewrite as a subprocedure, and its result is shown in Figure 3b. We treat *message* as if it were the queried relation, and apply MiM Rewrite to *session* as if it participated in

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

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 [8]. In [8], we prove that our example generalizes via the following corollary to Theorem 3.8.

COROLLARY 4.3. *Session Rewrite is query preserving and path-restricted for LR-LMR programs.*

4.2 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 [18]. Another is to maintain routes only to a few resource-rich proxies that in turn maintain many routes [13]. Our final rewrite, *Routing Rewrite*, exposes these options: it can reassign routing state to packets, proxies or some mixture of the two 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.

Routing Rewrite is applied in the same way as Session Rewrite except that the answer rules are set to a's LR rules (such as $\mathcal{R}_1$ in Listing 2). Consequently, 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. This relation "self-traversal" 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 from Theorem 3.8.

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 correctness proof and final result of Routing Rewrite on `BasicProg` are shown in [8].

5. 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*: searching for the optimal strategy.

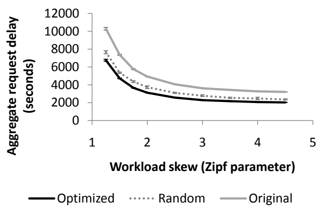
Inputs of *Decision Making* are network link costs and traffic profiles. Both the networking and database communities have extensively studied the problem of gathering such statistics [3, 15]. In the context of `netlog`, input data are all represented as relations. This information can be monitored regularly,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

Outputs of *Decision Making* are tuples for the relations *rendezvous* initialized by *Rewriting*. We implemented exhaustive search algorithms for each rewrite. For MiM Rewrite, we also adapted a greedy heuristic from the networking literature [23]. In principle, our rewrite-specific optimizations are replaceable by a general purpose dynamic programming optimizer, akin to those used widely by databases [25].

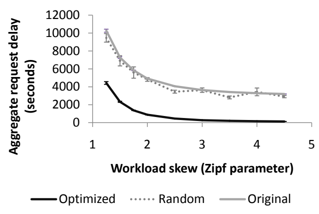
A benefit of the `netopt` architecture is that the analysis and rewrite to identify the optimization opportunity are distinct from the policy side of optimization. We have not focused on designing a better search algorithm for any specific scenario. Rather, we adopt an extensible framework which allows for the automated application of specific algorithms as appropriate [12]. This permits users to drop in custom optimizers that best suite the task at hand.

6. PROTOTYPE EVALUATION

We built a prototype `netopt` system that performs *Analysis*, *Rewriting* and *Decision Making*. The implementation uses Evita Raced, an extensible database optimizer [12], and the resulting programs run on the declarative sensornet platform DSN [10]. Our prototype still requires some user-assistance to link together the three steps. We evaluated the `netopt` prototype in the WSN settings discussed in Section 1.1. We tested on the Motelab testbed [21], as well as in simulation and on the Emulab testbed [14]. While not wireless,



(a) Concentrated provisioning



(b) Dispersed provisioning

Figure 4: Event distribution performance under varying provisioning layouts and workloads.

Emulab networks provided greater control over network topology than sensornet testbeds. We artificially limited Emulab hosts to resemble sensornet microservers. To evaluate declarative programs on Emulab hosts, we used P2 [19].

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 event distribution, we consider event notification delay. For other settings, we consider energy usage. In all settings, we see optimized programs outperforming unoptimized original programs. In event distribution, delay is decreased by as much as three orders of magnitude. In other settings, radio operations are decreased by as much as one order of magnitude. The overhead of optimization is a manageable increase in memory footprint for the installed program, as we will see in Section 6.5.

6.1 Event Distribution Rendezvous

The goal of event distribution is to decrease event notification delay of subscriber requests while working within topology constraints. We tested by simulation and on Emulab. For simulation, we randomly generated a 200 node topology. Some nodes are selected as event publishers and others as event subscribers. Publishers and subscribers are placed at nodes of low edge degree. While single base station scenarios (where all subscribers originate at the base) are common, we are observing that more dynamic distributed sensing-based actuation scenarios are emerging, where sensors trigger remote actuators, which may in turn trigger other sensors [9]; these are the types of configurations that we consider. There were 150 unique events. Each subscriber expressed a weighted Zipfian demand for each event. To experiment against varying workloads, we varied the skew of the Zipfian distribution.

Each node was also assigned an amount of available buffer space intended to reflect the capabilities of the underlying hardware. This buffer space is intended for caching of events from event publishers and interest notifications from event subscribers – a node needs enough marginal buffer space to serve as rendezvous for a publisher and subscriber pair. We considered two buffer provisioning layouts that seem to mirror practice. The first layout, Dispersed, has each node with an equal amount of buffer space. The second, Concentrated, has a few better-provisioned microserver-class nodes alongside resource-constrained nodes. Well-connected nodes were favored to receive available buffer space. The amount of aggregate buffer space was the same in both cases.

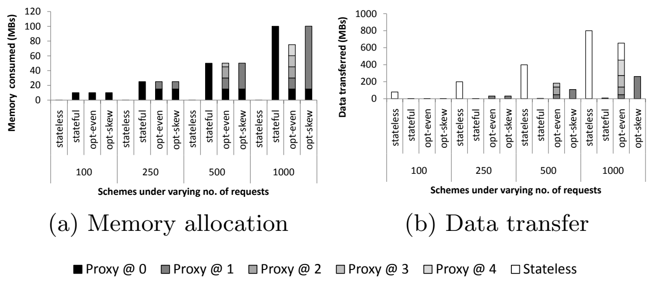
We experimented with four rendezvous selection schemes. The first, Original scheme, consisted of `BasicProg` in Listing 1 in which all subscriber requests go directly to the publishers with naive rendezvous. The remaining three schemes all used the rewritten `BasicProg` produced by MiM Rewrite in Listing 3. They differed in the *Decision Making* scheme employed. From the standpoint of the rewritten `BasicProg`, each scheme fed in its own *rendezvous* relation. The second, Random scheme, consisted of randomly assigning event types to available buffers. Here, resources were fully utilized, but the workload is not considered during assignment. The third, Optimized scheme, consisted of assigning content items to available buffers such that subscriber requests are serviced with lowest cost. The scheme used a greedy heuristic (by order of demand weight) for this assignment since the optimal assignment is known to be in NP-Complete. 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 4 shows the results of Original, Random and Optimized schemes under varying workloads and resources. Under the Concentrated buffer layout in Figure 4a, Random and Optimized performed 1.3-1.4× and 1.5-1.6× better than Original respectively as the workload varies from slightly skewed to highly skewed. Under the Dispersed layout in Figure 4b, Random and Optimized perform 0.95-1.2× and 2.3-24.4× better than Original respectively.

We also ran the same experiments on ten node Emulab networks. Random and Optimized outperformed Original by 1.5-1.9× and 2.8-3.3× with Concentrated, and by 1.2-2.3× and 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 subscriber workload and network resources.

6.2 Proxy Selection for a Single Service

We next tested the Session Rewrite, focusing on the



(a) Memory allocation (b) Data transfer

Figure 5: Single service proxy selection

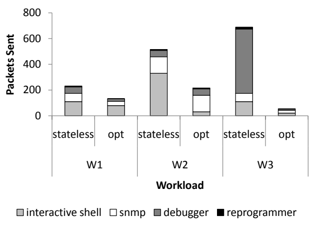


Figure 6: Multiple service proxy selection

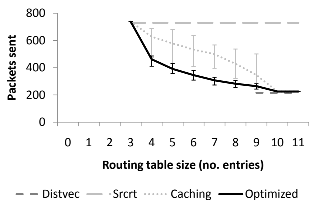


Figure 7: Routing proxy selection

case of a single microserver running a single generic networked service. We used a five hop (linear) Emulab network with node four making requests to node zero via nodes three, two and one. The linear network allowed us to isolate the study to the hop distance, and exclude fan-out considerations. The workload was varied from 100 to 1000 concurrent requests, which very well could be the case for a microserver with many connected sensors. Each request required 1Kb of session state. Two buffer provisioning layouts were used, Even and Skew. In Even, each node was allotted session buffer space of 15Mb, which was meant to represent prime main memory. In Skew, Node One was allotted 100MB for session state, whereas the other nodes were allotted 15MB. The Skew layout models a scenario in which a resource rich proxy is located close to the server.

The optimization objective was to minimize the total data transfer while serving all requests, since radio operations are often the most power-intensive activity. Three schemes were compared. In the first, Stateful, all session state was allocated at Node 0, regardless of whether the node buffer constraint was surpassed. This corresponds to the naive case of Figure 3a. In the second scheme, Stateless, all session state was packaged in request and response messages. A minimal amount of buffer was allocated at Node 0 to service these stateless requests. In the third scheme, Optimized, session state was assigned to proxies so as to minimize the total data transfer. This scheme tended to use as much buffer available at proxies closer to the server, Node 0, before using buffer further from the server. The Optimized scheme ran the program shown pictorially in Figure 3b.

Figure 5 shows the memory allocation and data transfer of each scheme under varying numbers of requests and buffer layouts. As expected, Stateless maintained an almost negligible amount of buffer usage across all nodes regardless of the number of requests, while its amount of data transfer grew very rapidly since it had to package all of its request state in packets. Conversely, as seen in Figure 5a, buffer usage under Stateful at Node 0 scaled 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 remained low even with many requests. Neither Stateful nor Stateless took advantage of the potential to use other nodes as proxies in the network, and therefore did not act differently when buffer layouts change.

The Optimized scheme is able to take into account varying buffer layouts. In Figure 5a, the "opt-even" and "opt-skew" labels show the resulting memory allocation on each node when the Session Rewrite optimizes against Even and Skew layouts respectively. At 100 requests when the buffer limit is not yet reached, Optimized behaves just like Stateful. At higher request counts, the constraint is respected by the Optimized scheme, and buffer is allocated from neighboring proxies rather than at node zero. Each segment of the stacked bars in Figure 5b 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 buffer constraints). This hybrid of stateless and stateful is a compromise when buffer constraints are present. Furthermore, the optimizer is able take advantage of the resource-rich proxy in the Skew layout, and transfer lower amounts of data (by using Node One) when buffer limits become an issue at higher request counts. The use of proxies does come at a cost: two bytes for both the buffer and in each packet transfer are needed for the join parameters which relink a request with its session state at the proxy.

6.3 Proxy Selection for Multiple Services

Next, we measured the Session Rewrite in a case with multiple services competing for the same buffer space. The optimizer attempts to minimize packets sent and received. 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 [17, 33, 30, 6]. 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.

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 a scenario in which the main application consumed 8Kb. Given the mote platform we were using, this left 2Kb main memory for our desired services [22].

We deployed Stateful, Stateless and Optimized programs on the Motelab testbed using DSN [10]. 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 20B 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 varied distributions 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 are shown in Figures 6. 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 6). Optimized allocates the most frequently called services' session state within the nodes' memory constraints, thus lowering the amount of packet state necessary.

6.4 Proxy Selection for Routing State

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. The buffer is constrained such that nodes only have space for a limited number of routing entries, varying from three to eleven. A typical sensornet routing services may only contain four entries [1]. The base station sends to nine destinations located four hops away in the network according to a Zipfian distribution.

Figures 7 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. 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.

6.5 Optimization Overhead

The overhead of optimization is primarily an increase in program rule count, resulting in larger memory footprints of optimized programs. Table 1 shows the optimized programs' memory usage when programs are compiled with DSN. For most cases, the increase is manageable – in the 8% to 63% range. The outlier is Routing where the rewrite meta-application is more complex and produces many more rules than the original program. In all cases, the programs fit comfortably on the target platform [22].

Table 1: Optimization Overhead in KB

Optimization	ROM	% Inc.	RAM	% Inc.
Rendezvous	25.2	13	3.4	63
Session	27.4	8	4.9	44
Routing	32.1	36	6.5	185

7. RELATED WORK

Related work stems from both networking and databases. Prior work in network protocol optimization generally focuses on packet processing performance on the single node, usually by adapting techniques from general compiler optimization to increase single-node packet processing performance: inlining, outlining, code cloning, rearranging branches, and IPC to function call conversion [7, 4] . On the other hand, our focus is automated multi-node protocol optimization.

Several efforts have attempted to enable greater network flexibility. Active networks research moved aggressively to introduce greater programmability into networks [32]. 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 [28]. Unlike our work, i3 does not aim to optimize their selection from program source.

The 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. System R popularized the ideas of optimizing join ordering with respect to disk IO, CPU, and table statistics [25]. Like [29], the current work fundamentally adopts the same optimization framework, extended to the networked setting. We significantly broaden the scope of what can be reordered, and thus what reordering is capable of by viewing "application data" and "networking data" under the same lens.

In the past, deductive database query optimization focused on combining "push" with "pull" query processing [31]. The main result was the Magic Sets algorithm that transforms programs to take advantage of the benefits of pull processing while executing in a push context [5]. The work of [20] extended this to the networked setting, specifically applying an entirely pull processing approach to the example of routing as in Section 4.2. 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 [31]. However, our experience indicates that LR is the most common recursion, especially in networking. This also echoes the remarks of [31] for traditional Datalog.

8. CONCLUSION

As sensornet workloads and resources continue to diversify, one-size-fits-all 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. This work naturally leads to further opportunities such as dynamic reoptimization, application to non-`netlog` programs, and new optimizations within the optimization architecture. Our initial study indicates that under a variety of sensornet settings, an informed optimizer can choose program executions that are much better than that of the original source program.

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

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

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

[7] 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.

[8] D. Chu and J. Hellerstein. Automating rendezvous and proxy selection. Technical Report UCB/EECS-2008-84, EECS Dept., UC Berkeley, 2008.

[9] D. Chu, F. Zhao, J. Liu, and M. Goraczko. Que: A Sensor Network Rapid Prototyping Tool with Application Experiences from a Data Center Deployment. In *EWSN08*.

[10] 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.

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

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

[13] C. Ee, S. Ratnasamay, and S. Shenker. Practical data-centric storage. In *NSDI*, 2006.

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

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

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

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

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

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

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

[24] R. Ramakrishnan and J. D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1993.

[25] 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.

[26] A. Sherman, P. A. Lisiecki, A. Berkheimer, and J. Wein. Acms: The akamai configuration management system. In *NSDI*, 2005.

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

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

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

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

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

[32] D. Wetherall. Active network vision and reality. In *SOSP*, 1999.

[33] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: a comprehensive source-level wsn debugger. In *SENSYS*, 2007.