The Case for a Unified Extensible Data-centric Mobility Infrastructure

Yun Mao
University of Pennsylvania
maoy@cis.upenn.edu

Boon Thau Loo
University of Pennsylvania
boonloo@cis.upenn.edu

Zachary Ives
University of Pennsylvania
zives@cis.upenn.edu

Jonathan M. Smith
University of Pennsylvania
jms@cis.upenn.edu

ABSTRACT

We present a unified, extensible data-centric mobility infrastructure based on *declarative networks* and *composable distributed views* over network, router, and host state. Declarative networks are a recent innovation for building extensible network architectures using declarative languages. The data-centric approach both improves flexibility over existing solutions, and is extensible to meet the demands of future mobile applications and services. We demonstrate the flexibility of distributed queries used in declarative networks by specifying and implementing mobile services, e.g., overlay-based solutions for host mobility, customizable routing, service discovery and composition, and location-based services. A prototype based on the P2 declarative networking system has been implemented, with which we evaluated two overlay-based mobility schemes (ROAM and DHARMA).

1. INTRODUCTION

The Internet's role is changing dramatically, from a means of connecting together PCs and servers to a ubiquitous, over-the-air communications medium interconnecting mobile personal devices, environmental sensors, and Web services. Broadband wireless MANs are being deployed in many cities, and emerging technologies (e.g., WiMAX) are built into many communications devices, as well as laptops.

As new services (voice, video, emergency response, *etc.*) are being deployed on the wireless infrastructure, there have been increased demands on the Internet for efficient routing among mobile and wired nodes, location of proximity-based services, and wide-area service discovery and composition. To address these demands, a variety of special-purpose protocols and overlay networks [11, 16] have been developed, focusing on flexible naming and addressing of mobile hosts, locating proxies and home agents of mobile hosts, and supporting location-based services.

In this paper, we present a new point in this design space that aims to achieve extensibility in mobility infrastructures based on the application of database techniques to the networking domain. We propose a unified, extensible data-centric mobility infrastructure that allows for (1) extensible capabilities for *logical* naming of data in the form of *views*[1], (2) unified declarative queries for *distributed data management and state acquisition*. We argue that our approach will make it much easier to combine existing services, choose among alternatives, and deploy new services.

Logical-physical data model separation and distributed data acquisition and transformation are the cornerstones of the modern database field [5, 10, 4]. Hence, we propose that a *declarative*, database-inspired mobility infrastructure using *query languages* is an ideal interface to these capabilities. Declarative languages provide both optimization and strong static verification possibilities, and declarative query languages focus on distributed data acquisition and transformation. Moreover, because such languages allow for very general definitions and are concise, they are much more extensible over time. Views can be composed, new queries and requests can be deployed, and new optimization or processing techniques can be invented and deployed without breaking compatibility. Such properties differentiate our approach from work on Active Networks [15], which typically used general programming models and only had a limited treatment of distributed coordination and data acquisition.

Our mobility infrastructure uses *declarative networks* [9, 8] to build extensible network architectures that achieve a good balance of flexibility, performance and safety. Declarative networks are specified using *Network Datalog* (*NDlog*), which is a distributed recursive query language used for querying network graphs stored in databases. *NDlog* queries are executed using a distributed query processor to implement the network protocols, and continuously maintained as distributed views over existing network and host state.

Declarative queries such as *NDlog* are a natural and compact way to implement a variety of routing protocols and overlay networks. For example, traditional routing protocols can be expressed in a few lines of code [9], and the Chord [14] distributed hash table in 47 lines of code [8]. When compiled and executed, these declarative networks perform efficiently relative to imperative implementations. The orders of magnitude reduction in code size significantly increases programmer productivity. Moreover, these declarative specifications

[1]In database terminology, a view is a virtual or logical table composed of the result set of a query.

MobiArch'07, August 27–31, 2007, Kyoto, Japan.

allow mobile services to be easily composed, and added to the infrastructure. While declarative networking concepts apply generally to a variety of networked environments, the wireless access and network mobility environment presents a compelling use case for these ideas. Examples include:

Flexible addressing and naming: Many mobile addressing and naming schemes are currently supported through indirection, which maps names to addresses in a location-sensitive way. We can generalize these capabilities by composing and then customizing these indirection requests as declarative queries, which can also take into account other types of relevant data, e.g., subscriber status or system load. These queries are maintained continuously as distributed views. Mobile hosts can then periodically query these views in order to select proxies as they move in the network.

Customizable routing: Mobile environments greatly increase the heterogeneity and complexity of networks. The need for better management and diagnosis tools, as well as adaptive and self-healing routing protocols, requires distributed acquisition of network state and data. Declarative routing [9] protocols are a natural fit in this environment, and can be used to specify additional constraints for session requirements and network state.

Declarative service specifications: Service discovery and composition will become increasingly necessary to support transcoders and location-based services. This is naturally supported by storing service descriptions as relational tables, which are queried and composed by mobile hosts.

Security: With the proliferation of mobile devices and wireless communication networks, security and intrusion detection are increasingly important. Such services require correlation of state and activity distributed across a multitude of subnetworks and hosts. A variety of network monitoring services can be deployed as distributed queries, and data-centric views offer a natural way for enforcing access control [1]. Our unified data-centric framework permits the easy integration of declarative networks with network monitoring queries. Moreover, static analysis techniques [7] can be applied to ensure that only queries that terminate are permitted.

Contributions and Organization: Section 2 presents an architectural overview of the extensible data-centric mobility infrastructure based on the P2 declarative networking system, and introduces the *NDlog* query language. Section 3 shows the flexibility of the mobility infrastructure in specifying flexible proxy location, customizable routing, and service discovery and composition. Section 4 demonstrates the performance of our approach, showing initial evaluation results on the Emulab testbed for two overlay-based mobility schemes (ROAM [16], previously implemented over the Internet Indirection Infrastructure (i3) [13], and DHARMA [11]) implemented using the P2 System. Section 5 concludes the paper with a discussion of future research directions.

2. SYSTEM OVERVIEW

Figure 1 illustrates the proposed mobility infrastructure at a conceptual level. The nodes in the mobility infrastructure can either be routers or overlay nodes in an existing communication infrastructure. These nodes provide a variety of services on behalf of mobile hosts, including discovery

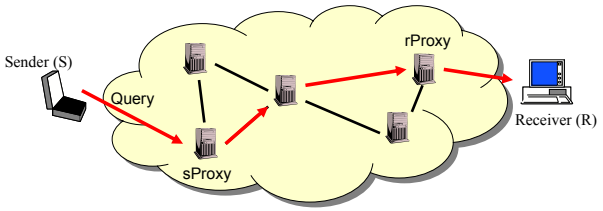

Figure 1: An overview of the mobility infrastructure.

and assignment of nearby nodes as home agents (or *proxies*) for routing data, adaptive routing among different mobile hosts based on session requirements, location-based services, service discovery and composition, and network monitoring within the infrastructure.

These services are supported in our system as distributed queries, specified in the *NDlog* query language used in declarative networking. *NDlog* queries can either be initiated by a mobile host (e.g. for locating nearby services or home agents), or executed continuously at the infrastructural-level (e.g. maintaining the network state of an overlay that supports mobility, tracking the location of mobile hosts, monitoring of network state, etc.).

As an example, in Figure 1, the mobile host (S) issues a query to discover a nearby proxy node (*sProxy*), and a route is established (via the execution of a declarative routing [9] query) to the proxy on the receiving side (*rProxy*). The route can be computed based on the bandwidth requirements of the session, and maintained for the duration of the session.

2.1 Infrastructure Node

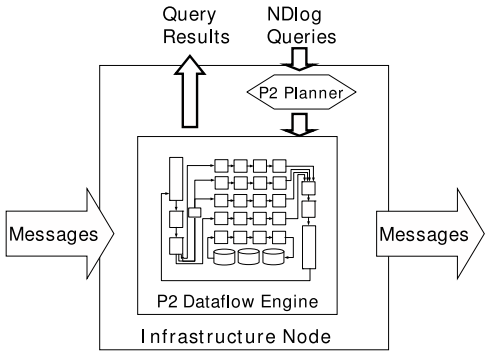

Figure 2: Components of an infrastructure node.

The mobility infrastructure utilizes the P2 declarative networking system to process *NDlog* queries to implement the mobile services. The P2 system runs on a set of *infrastructure nodes*, and queries can execute in distributed fashion across these nodes. Figure 2 shows the components of a single infrastructure node based on the fully-distributed execution model. We provide a high-level description of the P2 system[2]. There are two main components in P2: the *planner*, and the *dataflow engine*. The P2 planner takes as in-

[2]Interested readers should see references [9, 8] for details on query planning, optimizations, and the dataflow framework.

put *NDlog* queries, which are compiled into execution plans (dataflows), and then executed using P2's dataflow engine. Unlike traditional database execution plans, P2 dataflows are suitable for implementing network protocols, and share a similar execution model with the Click modular router [6], which consists of elements that are connected together to implement a variety of network and flow control components. In addition, P2 elements include database operators (such as joins, aggregation, selections, and projects) that are directly generated from the queries.

Each local dataflow participates in a global, *distributed* dataflow, with messages flowing among dataflows executed at different nodes, resulting in updates to local tables, or query results that are returned to the mobile hosts that issued the queries. The local tables store the state of intermediate and computed query results, which include the network state of various network protocols in support of mobility, stored state of mobile hosts, and location-based service descriptions. The distributed dataflows implement the operations of various network protocols in support of mobility. The flow of messages entering and leaving the dataflow constitute the network messages generated during query execution.

2.2 NDlog Query Language

Datalog is a recursive query language primarily used in the database community for querying graph structures. We provide a short review of Datalog, following the conventions in Ramakrishnan and Ullman's survey [12]. A Datalog program consists of a set of declarative *rules* and a query[3].

A Datalog *rule* has the form `p :- q1, q2, ..., qn.`, which can be read informally as `q1` and `q2` and `...` and `qn` implies `p`". Here, `p` is the *head* of the rule, and `q1, q2,...,qn` is a list of *literals* that constitutes the *body* of the rule. Literals are either *predicates* applied to *fields* (variables and constants), or boolean expressions that involve function symbols (including arithmetic) applied to fields. We sometimes refer to these fields as *attributes*. The rules can refer to each other in a cyclic fashion to express recursion. The order in which the rules are presented in a program is semantically immaterial. The commas that separate the predicates in a rule are logical conjunctions (*AND*); the order in which predicates appear in a rule body also has no semantic significance (though implementations typically employ a left-to-right execution strategy). The query specifies the output of interest. We can convert the query into a view simply by giving it a name; other queries can use this view as if it were a table.

The predicates in the body and head of traditional Datalog rules are relations, and we will refer to them interchangeably as predicates, relations, or tables. Each relation has a *primary key*, which consists of a set of fields that uniquely identifies each tuple within the relation. In the absence of other information, the primary key is the full set of fields in the relation.

The names of predicates, function symbols, and constants begin with a lower-case letter, while variable names begin with an upper-case letter. Most implementations of Datalog enhance it with a limited set of function calls (which start with "`f_`" in our syntax), including boolean predicates, arithmetic computations, and simple list operations. Aggregate constructs are represented as functions with field variables within angle brackets (`<>`). We will not consider negated predicates since they are not supported by *NDlog*.

Network Datalog (*NDlog*) is a distributed variant of traditional Datalog, primarily designed for expressing distributed recursive computations common in network protocols. We illustrate *NDlog* using a simple example of two rules that computes all-pairs of reachable nodes:

```
r1 reachable(@S,D) :- link(@S,D).
r2 reachable(@S,D) :- link(@S,Z), reachable(@Z,D).
Query reachable(@S,D).
```

The rules `r1` and `r2` specify a distributed transitive closure computation, where rule `r1` computes all pairs of nodes reachable within a single hop from all input links, and rule `r2` expresses that "if there is a link from `S` to `Z`, and `Z` can reach `D`, then `S` can reach `D`." By modifying this simple example, we can construct more complex routing protocols, such as the distance vector and path vector routing protocols. See [9] for more details.

NDlog supports a *location specifier* in each predicate, expressed with `@` symbol followed by an attribute. This attribute is used to denote the source location of each corresponding tuple. For example, all `reachable` and `link` tuples are stored based on the `@S` address field. The output of interest is the set of all `reachable(@S,D)` tuples, representing reachable pairs of nodes from `S` to `D`. In addition, we can bind the query output using constants, e.g. "`Query reachable(@a,D)`" outputs nodes reachable from `a`.

3. QUERIES FOR MOBILITY

In this section, we provide three example classes of useful queries in support of mobility: *proxy location*, *customizable routing*, and *service discovery and composition*. This is not intended to be an exhaustive coverage of the possibilities of our proposal, but rather an illustration of the ease with which *NDlog* queries can be used for implementing mobile services that can be easily composed and enhanced for various aspects of mobility.

3.1 Proxy Location

In our first example, a mobile host issues an *NDlog* query to locate a nearby infrastructure node that serves as its proxy node either for routing data, or for providing support for location-based services. The examples are based on ROAM [16] and DHARMA [11].

3.1.1 ROAM

ROAM is built on top of i3 [13], a rendezvous-based overlay network that provides a level of indirection between senders and receivers. In i3, instead of explicitly sending a packet to a destination, each packet is associated with an identifier; this identifier is then used by the receiver to achieve delivery of the packet. This indirection mechanism is implemented with the use of *triggers*. In our system, each `trigger(@PI,P,NI)` condition is encoded in a tuple, which is stored at the infrastructure node `PI`; it specifies to route messages on behalf of mobile host `NI` with identifier `P`. A mobile host with identifier `P` and address `NI` needs to update its trigger only when its address changes. The following rules `i1-i7` are issued by a mobile host for selecting the closest infrastructure node as its proxy.

[3]Since these programs are commonly called *"recursive queries"* in the database literature, we will use the term "query" and "program" interchangeably when we refer to a Datalog program.

```
#define SIZE 5
#define S_RATE 60
i1 randomKeys(@NI,K) :- periodic(@NI,E,0,SIZE),
                        K=f_randID().
i2 lookup(@LI,K,NI,E) :- periodic(@NI,E1,S_RATE),
                         randomKeys(@NI,K),
                         landmark(@NI, LI),
                         E=f_rand().
i3 pendingPing(@NI,R,RI,E1,T) :-
          lookupResults(@NI,K,R,RI,E), T=f_now().
i4 pingReq(@RI,NI,E) :- lookupResults(@NI,K,R,RI,E).
i5 pingRTT(@NI,R,RI,RTT) :-
          pingResp(@NI,SI,E),
          pendingPing(@NI,R,RI,E,T), RTT=f_now()-T.
i6 leastRTT(@NI,E,min<RTT>) :-
          periodic(@NI,E,S_RATE),
          pingRTT(@NI,R,RI,RTT).
i7 proxy(@NI,R,RI) :- lowestRTT(@NI,E,RTT),
                      pingRTT(@NI,R,RI,RTT).
Query proxy(@NI,R,RI).
```

The i3 overlay utilizes a DHT to provide the mapping from identifiers to hosts. Our rules assume that the declarative overlay implementation of the Chord DHT (referred to as *P2-Chord* [8]) is executed on all infrastructure nodes.

Rule `i1` periodically[4] generates `SIZE` number of random keys stored in `randomKeys` table using the built-in function `f_randID` that will return a 160-bit random identifier. Every `S_rate` seconds, rule `i2` generates `lookup(@LI,K,NI,E)` requests from the mobile host `NI` to its landmark node `LI`, one for each random key `K` generated in rule `i1`. A unique event identifier `E` is associated with each lookup. Based on the lookup results obtained from P2-Chord, rules `i3-i4` will result generate `pingReq` request tuples to all infrastructure nodes in the lookup results.

Each infrastructure node responds to the ping requests with `pingResp` tuples that are sent back to the mobile host. Upon receiving the `pingResp`, rule `i5` computes the round-trip-time (RTT) from the mobile host to the infrastructure node. All the computed RTTs are then used in rule `i6-i7` for selecting the closest proxy node. The selected proxy with address `RI` and identifier `R` (denoted by `proxy(@NI,R,RI)` tuple) will be maintained at the mobile host `NI`. With an additional rule "`i8 trigger(@RI,R,NI) :- proxy(@NI,R,RI)`", the mobile host `NI` can insert a trigger tuple at its selected proxy node `RI` storing its address and identifier. All packets to `NI` are then routed to the indirection node `RI` which then forwards them to `NI` via this trigger.

3.1.2 DHARMA

DHARMA [11] can also be used to select a nearby proxy node. Unlike ROAM's DHT, DHARMA uses a number of designated *portal servers*, nodes that store information on other nodes in the infrastructure.

```
#define R_RATE 5
d1 agentMsg(@PI,NI) :- periodic(@NI,E,R_RATE),
                       portal(@NI,PI).
d2 pingResp(@RI,NI,E) :- pingReq(@NI,RI,E).
```

We consider a simple example with one portal server `PI`, stored on every infrastructure node `NI` as a `portal(@NI,PI)` tuple. Rule `d1` is executed at all infrastructure nodes, and will result in the generation of periodic heartbeats (`agentMsg(@PI,NI)` tuples) to the portal server `PI`. Rule `d2` generates a `pingResp` in response to a ping request.

```
d3 agentList(@PI,NI) :- agentMsg(@PI,NI).
d4 agentCandidates(@MI,AI,E) :-
          requestProxy(@PI,MI,S,E),
          agentList(@PI,AI), f_coinFlip(S)=1.
```

Rules `d3-d4` are executed on the portal server. Rule `d3` stores all incoming `agentMsg(@PI,NI)` tuples in a `agentList(@PI,NI)` table, hence maintaining the list of all candidate infrastructure nodes `NI`. These tuples will timeout unless the respective infrastructure nodes periodically refresh their entries via regular `agentMsg` messages. Rule `d4` is generated in response to a mobile host request for candidate proxies (via a `requestProxy` tuple).

```
#define P_INTERVAL 5
#define S_RATE 0.2
d5 requestProxy(@PI,NI,S_RATE,E) :-
         periodic(@NI,E,P_INTERVAL),
         portal(@NI,PI).
d6 pendingPing(@NI,AI,E,T) :-
         agentCandidates(@NI,AI,E), T=f_now().
d7 pingReq(@AI, NI, E) :- agentCandidates(@NI,AI,E).
d8 pingRTT(@NI,SI,RTT) :- pingResp(@NI,SI,E),
         pendingPing(@NI,SI,E,T), RTT=f_now()-T.
d9 leastRTT(@NI,E,min<RTT>) :-
         periodic(@NI,E,P_INTERVAL),
         pingRTT(@NI,RI,RTT).
d10 leastRTTNode(@NI,RI,RTT) :- leastRTT(@NI,E,RTT),
         pingRTT(@NI,RI,RTT).
d11 proxy(@NI,RI) :- lowestRTT(@NI,E,RTT),
                     pingRTT(@NI,RI,RTT).
Query proxy(@NI,RI).
```

The above rules `d5-d11` are executed by a mobile host seeking to locate a nearby proxy node. Rule `d5` results in the mobile host periodically generating a `requestProxy` tuple to the portal server, which will return 20% (determined by `S_RATE`) of the nodes from its `agentList` table as potential proxy candidates. Similar to the earlier ROAM rules, `d6-d11` computes the closest proxy node `RI`, which is maintained at the mobile host `NI`.

3.1.3 Flexible Proxy Selection

The *NDlog* query language enables higher-level concepts to be easily encoded by making minor modifications to the above rules. This enables user-customizable proxy selections. For example, by replacing `min` with `min-k`, we can select the *top-k* nodes to get multiple proxies per mobile host. We can also adopt different criteria: instead of selecting the closest RTT node, we can select the least loaded node as long as it is within a RTT bound. We can also limit our proxy selection to nodes that provide certain services within their location (e.g. transcoding services described in Section 3.3).

3.2 Customizable Routing

Our next example shows a customizable version of the basic path vector protocol [9]. The query computes the best paths among all infrastructure nodes. The query takes as input `link(@S,D,C)` tuples, where each link from node `S` to node `D` denotes connectivity between two infrastructure nodes; messages can be routed from `S` to `D` at cost `C`.

[4]The `periodic(@N,E,T,K)` event relation is a built-in relation that represents a stream of event tuples generated at node `N` every `T` seconds (up to an optional `K` times) with a random event identifier `E`. If `K` is omitted, the stream is generated infinitely.

```
bp1 path(@S,D,D,P,C) :- link(@S,D,C), P=f_init(S,D).
bp2 path(@S,D,Z,P,C) :- link(@S,Z,C1),
                        path(@Z,D,Z2,P2,C2),
                        C=f_compute(C1,C2),
                        P=f_concatPath(S,P2).
bp3 bestPathCost(@S,D,AGG<C>) :- path(@S,D,Z,P,C).
bp4 bestPath(@S,D,P,C) :- bestPathCost(@S,D,C),
                          path(@S,D,Z,P,C).
Query bestPath(@S,D,P,C).
```

Rules `bp1` and `bp2` compute all possible paths, and rules `bp3` and `bp4` compute all-pairs best paths, which are stored as `bestPath` tuples at each source node `S` for source routing. We have left the aggregation function `AGG` unspecified. By changing `AGG` and the function `f_compute` used for computing the path cost C, the above query can generate best paths based on any metric including link latency, loss rates, available bandwidth and node load. For example, if the query is used for computing the shortest paths, `f_sum` is the appropriate replacement for `f_compute`, and `min` is the replacement for `AGG`. The above query can be further restricted by the current sender and receiver proxies of the communicating devices, and the routes between these two proxies is maintained as a continuous query, and adapted based on link updates.

We can extend the query by adding constraints based on the session requirements, by introducing an additional `session` predicate to the rules above. For example, we can restrict the set of paths to those with costs below a loss or latency threshold K by adding a `session(@S,K)` predicate, and a constraint `C<K` to the rules `bp1` and `bp2`.

3.3 Service Discovery and Composition

The proxy location query described in Section 3.1 is an instance of service discovery, in which a nearby routing proxy is located. Once a proxy node is identified, a mobile host can issue additional queries via its proxy to locate desired resources within its vicinity. In addition, the use of a declarative framework eases the composition of services. For example, one can query for multiple services within the infrastructure, and then construct a path (either an explicit path presented Section 3.2, or a series of triggers supported by i3) along all intermediate service points.

In our final example, we build upon the earlier ROAM example, to demonstrate service discovery and composition with the use of i3 triggers. Here a sender `SI` performs a discovery of a transcoder, and then forwards all packets to the transcoder before being delivered to their receivers `RI`. Our example is presented with flexibility and composability of our infrastructure in mind. While our example is based on i3 and ROAM, our infrastructure does not preclude supporting other discovery and composition mechanisms.

```
t1 leastLoad(@PI,SI,min<L>) :-  proxy(@SI,P,PI),
          transcoders(@PI,TI,TID,L).
t2 bestTranscoder(@SI,TI,TID) :-
          transcoders(@PI,TI,TID,L),
          leastLoad(@PI,SI,L).
Query bestTranscoder(@SI,TI,TID).
```

Each transcoder `TI` inserts a trigger, stored as a `trigger(@PI,TID,TI)` tuple at the infrastructure node `PI` that owns the identifier `TID`. These triggers are further registered in the `transcoders` table of nearby proxy nodes. The rules `t1-t2` are used by a sender `SI` to locate a least loaded transcoder `TI` with identifier `TID` registered at `S`'s proxy `PI`.

When a sender `SI` wishes to send a packet to a mobile host `NI` supported by a trigger with identifier `RID`, a series of identifiers `(TID,RID)` is required to first forward each packet to the transcoder (`TID`), which then reroutes the packet to the mobile host (`RID`). We omit the (few) *NDlog* rules for forwarding with triggers. The main takeaway is that i3's service composition mechanism via multiple triggers can be easily supported by our infrastructure. In addition, we can use the declarative interface to query and locate the triggers themselves during service discovery.

As a further enhancement, we can enhance existing queries to support *late binding* [2]. An *NDlog* query can be sent along with each packet, which will be routed based on the service requests indicated in the *NDlog* query. Support for late bindings require additional *NDlog* rules for publishing and propagating service descriptions among infrastructure nodes to create routing tables based on service attributes. This provides a flexible data delivery mechanism that allows applications to track rapid change (e.g., user or network mobility) and support changing service updates.

4. PRELIMINARY EVALUATION

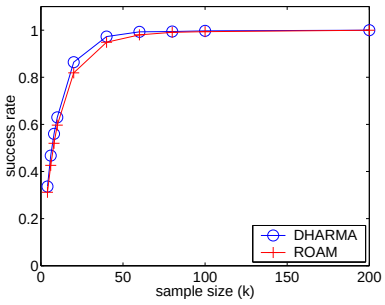

Figure 3: Success rate vs sample size for proxy location using DHARMA and ROAM. The success rate is the probability of a mobile host picking a proxy that is within the same routing domain.

In this section, we present a preliminary evaluation of our proposed infrastructure using the P2 system on the Emulab [3] testbed. We focus our evaluation on the proxy location queries (see Section 3.1) for ROAM and DHARMA. Our experimental setup on Emulab consists of 50 nodes organized into 11 routing domains interconnected in a star topology. On each Emulab node, we run 4 P2 processes, for 200 emulated nodes in total.

In our first experiment, we execute the rules of Section 3.1.2 that implement the DHARMA overlay network. We first randomly pick one node to be the portal server maintaining a list of current live home agents. The remaining nodes then periodically report their status to the portal server. In steady state, we randomly pick one of the Emulab nodes as a mobile host. This mobile host issues a query to the portal server, which triggers additional rules (see Section 3.1.2) to select k nodes as samples, and then locates the closest node (in terms of network latency) from these sample nodes as the proxy. If the chosen proxy is within the same routing domain as the mobile host, the proxy location "succeeds."

Figure 3 shows the success rate of DHARMA as the sample size increases. For any given sample size, we execute

the proxy selection query from 1000 randomly selected locations in the network. As expected, when the sample size increases, the success rate increases. When the sample size is as large as the number of infrastructure nodes (200), all nodes are sampled, and hence the closest proxy is guaranteed to be within the domain. Sampling only 10% of the nodes, DHARMA achieves a success rate of 85%.

In our second experiment, we execute P2-Chord [8] on all infrastructure nodes for proxy selection. After starting a 200-node Chord network, each mobile host executes the proxy selection queries (as described in Section 3.1.1) by repeatedly sampling the infrastructure nodes via Chord lookups. The closest node in network distance is then selected as the proxy. Figure 3 shows the corresponding success rate of ROAM as the sample size increases. As before, we iterate the proxy selection process from 1000 randomly selected network locations. ROAM's performance is roughly that of DHARMA. When the sample size is 20 nodes, the probability of picking a close proxy is 81%. ROAM has a slightly worse success rate than DHARMA for small k, because the node identifiers are not uniformly distributed for the small 200-node Chord network. As the number of Chord nodes increases, the identifier space will be more evenly distributed, and the performance of ROAM will approach DHARMA. ROAM has the advantage that it avoids the use of a centralized portal server.

The results on Emulab show that we can implement DHARMA and ROAM to perform effective location of proxies that map to nearby locations. As shown in Section 3.1, these specifications can be written in a few *NDlog* rules each, significantly easing the process of deploying new mobility-based solutions. Comparisons between DHARMA and ROAM are not the point of this paper, rather our experiences with two suggests that our infrastructure can be used to rapidly develop and deploy multiple concurrent mobility-based schemes.

5. CONCLUSIONS AND FUTURE WORK

We have presented the case for a unified extensible data-centric mobility infrastructure based on declarative networking. Using examples, we showed that the data-centric approach both improves flexibility and is extensible to meet the demands of future mobile applications and services.

This research is proceeding in several directions. First, we are planning a full-fledged implementation using the P2 system, and we will evaluate performance of the system on Emulab and Planetlab testbeds. Beyond proxy selection queries, we intend to support other aspects of mobility, including customizable routing among different mobile devices, declarative service discovery and composition, integrating distributed network monitoring queries with overlay-based solutions for mobility, enforcing view-based access control policies [1], and mobility prediction with prefetching and caching. We also intend to investigate the use of multi-query optimization techniques to share among multiple overlay-based mobility schemes that can be deployed concurrently in our system.

6. ACKNOWLEDGMENTS

This material is based upon work supported in part by NSF IIS-0477972 and IIS-0513778.

7. REFERENCES

[1] Abadi, M., and Loo, B. T. Towards a Language and System for Secure Networking. In *NetDB* (2007).

[2] Adjie-Winoto, W., Schwartz, E., Balakrishnan, H., and Lilley, J. The design and implementation of an intentional naming system. In *SOSP* (1999), pp. 186–201.

[3] Emulab. Network Emulation Testbed. http://www.emulab.net.

[4] Halevy, A. Y., Ives, Z. G., Suciu, D., and Tatarinov, I. Schema mediation in peer data management systems. In *Proceedings of IEEE Conference on Data Engineering* (2003).

[5] Huebsch, R., Hellerstein, J. M., Lanham, N., Loo, B. T., Shenker, S., and Stoica, I. Querying the Internet with PIER. In *Proceedings of VLDB Conference* (2003).

[6] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The Click Modular Router. *ACM Transactions on Computer Systems 18(3)* (2000), 263–297.

[7] Krishnamurthy, R., Ramakrishnan, R., and Shmueli, O. A Framework for Testing Safety and Effective Computability. *J. Comp. Sys. Sci. 52(1):100-124* (1996).

[8] Loo, B. T., Condie, T., Hellerstein, J. M., Maniatis, P., Roscoe, T., and Stoica, I. Implementing Declarative Overlays. In *ACM Symposium on Operating Systems Principles* (2005).

[9] Loo, B. T., Hellerstein, J. M., Stoica, I., and Ramakrishnan, R. Declarative Routing: Extensible Routing with Declarative Queries. In *Proceedings of ACM SIGCOMM Conference on Data Communication* (2005).

[10] Madden, S., Franklin, M. J., Hellerstein, J. M., and Hong, W. Design of an acquisitional query processor for sensor networks. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (2003), pp. 491–502.

[11] Mao, Y., Knutsson, B., Lu, H., and Smith, J. M. DHARMA: Distributed Home Agent for Robust Mobile Access. In *IEEE INFOCOM* (2005).

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

[13] Stoica, I., Adkins, D., Zhuang, S., Shenker, S., and Surana, S. Internet Indirection Infrastructure. In *Proceedings of ACM SIGCOMM Conference on Data Communication* (2002).

[14] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM Conference on Data Communication* (2001).

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

[16] Zhuang, S. Q., Lai, K., Stoica, I., Katz, R. H., and Shenker, S. Host Mobility using an Internet Indirection Infrastructure. In *ACM/Usenix Mobisys* (2003).