A^3: An Extensible Platform for Application-Aware Anonymity

Micah Sherr* Andrew Mao† William R. Marczak‡
Wenchao Zhou* Boon Thau Loo* Matt Blaze*

*University of Pennsylvania †Harvard University ‡University of California, Berkeley
{msherr,wenchaoz,boonloo,blaze}@cis.upenn.edu, mao@seas.harvard.edu, wrm@berkeley.edu

Abstract

This paper presents the design and implementation of Application-Aware Anonymity *(A^3), an extensible platform for deploying anonymity-based services on the Internet. A^3 allows applications to tailor their anonymity properties and performance characteristics according to specific communication requirements.*

*To support flexible path construction, A^3 exposes a declarative language (*A³Log*) that enables applications to compactly specify path selection and instantiation policies executed by a declarative networking engine. We demonstrate that our declarative language is sufficiently expressive to encode novel multi-metric performance constraints as well as existing relay selection algorithms employed by Tor and other anonymity systems, using only a few lines of concise code. We experimentally evaluate the A^3 system using a combination of trace-driven simulations and deployment on PlanetLab. Our experimental results demonstrate that A^3 can flexibly support a wide range of path selection and instantiation strategies at low performance overhead.*

1 Introduction

In the past decade, there has been intense research [9, 31, 30, 13, 39, 44, 32, 23, 38, 4, 10, 36, 40, 24] into designing systems that enable parties to communicate anonymously in the presence of eavesdroppers. Typically, these systems achieve anonymity by sending a message through a path of relays before delivering it to its final destination. Broadly speaking, recent innovations have improved *relay selection* [8, 40, 24, 36]) – choosing a path of relays to provide high anonymity and good performance – and *path instantiation* – establishing necessary state at each relay to enable anonymous communication.

Despite the proliferation of proposed techniques, we note that no *one-size-fits-all* anonymity system exists. The appropriate relay selection and path instantiation strategy can vary according to application requirements, performance characteristics, and additional constraints imposed by the underlying network. For example, in the context of relay selection, an anonymous video conferencing system may be willing to achieve weaker anonymity in exchange for a path that meets its high-bandwidth, low-latency performance demands. In contrast, an anonymous email system may require very strong anonymity guarantees while imposing no constraints on bandwidth or latency.

Similarly, several path instantiation approaches exist. Onion Routing [30] and Tor's *telescoping* scheme [9] build paths by recursively encrypting and shipping key material to their constituent nodes. The former constructs anonymous paths that have constant length over their lifetime, while the latter adds the ability to extend existing anonymous paths. On the other hand, the *Crowds* [31] approach relies on the network to make routing decisions on behalf of the source. Crowds is best suited for an environment where source routing is not available and intermediate relay nodes can be trusted with the identity of the receiver.

In this paper, we present the *Application-Aware Anonymity* (A^3) framework: an anonymity system that enables tradeoffs between anonymity and performance through highly customizable relay selection and path instantiation strategies. A^3 aims to support a wide range of anonymity-based networked services with different application-specific constraints. Applications can leverage A^3 in a *policy-driven* fashion by specifying path instantiation and relay selection techniques that meet their performance and anonymity requirements.

One important element of A^3 is the use of *declarative networking* [20, 19], a declarative logic-based frame-

work that can efficiently execute a high-level protocol specification using orders of magnitude less code than an imperative implementation. A^3 utilizes a declarative networking system as a policy engine for specifying and executing relay selection and path instantiation policies.

Our proposed A^{3}LOG declarative language extends previous declarative networking languages with constructs that are added specifically to enable the specification of anonymity systems. For example, we have integrated the ability to specify user-defined cryptographic primitives for secure communication. We have also adapted recently proposed extensions for declarative network composition [22] to enable us to develop reusable components ideal for specifying and customizing anonymous routing. We demonstrate how these extensions enable the concise expression of relay selection and path instantiation algorithms.

A^3 is sufficiently extensible to support both traditional *node-based* as well as recently proposed *link-based* [36, 35] relay selection strategies. Node-based strategies select relays with desirable node properties (usually bandwidth), whereas link-based strategies bias relay selection in favor of link characteristics such as latency, AS hop count, or jitter. We demonstrate that both link- and node-based relay selection strategies, including those used by Tor and other systems, can be concisely represented in a few lines of A^{3}LOG code. We also show how A^{3}LOG compactly encodes the path instantiation algorithms used by these systems. By providing a flexible framework for realizing both relay selection and path instantiation policies, A^3 enables the rapid development, deployment, and testing of both existing and novel anonymity protocols.

We experimentally evaluate the A^3 system through both trace-driven simulations and a deployment on PlanetLab. Our results demonstrate that the A^3 system can flexibly support a wide range of path selection and instantiation strategies at low performance overhead.

2 Related Work

To support diverse applications, the Internet uses a simple routing scheme in which packets are forwarded on a best-effort basis towards their intended destinations. The end-to-end (e2e) performance of Internet paths is dictated by policies enforced by routers along the path from source to sink. With the exception of fragmented portions of the Internet that support IP quality-of-service features, applications usually have little control over the performance aspects of their network connections.

An *overlay network* built on top of the Internet routing infrastructure can allow users to exercise greater control over the manner in which their messages are relayed, as forwarding can be based on application layer information. When combined with source-routing, these networks allow applications the ability to select paths that meet their specific requirements (e.g. RON [2] for robustness).

Overlay networks may also enable anonymous routing on the Internet. For example, Tor [9], Onion Routing [30], Crowds [31], Tarzan [13], Hordes [39], JAP [10], and MorphMix [32] (among many others) utilize application-layer overlay routing. These anonymity systems exploit two features of overlay networks: (i) the ability to obfuscate the addresses of the *initiator* (sender) and *responder* (receiver) while still providing reliable message delivery; and (ii) in some instances, the ability to produce anonymous paths that achieve some desirable property (usually high bandwidth) [9, 13, 40]. This paper is principally concerned with the latter aspect: *We provide mechanisms that allow anonymity systems to produce desirable paths*.

A large volume of existing literature examines methods for generating high performance anonymous paths. Tor [9, 8] attempts to achieve high bandwidth paths by imposing a probability distribution over the set of potential anonymous relays. The probability of a relay being selected is proportional to its advertised bandwidth. Murdoch and Watson have demonstrated that such a strategy delivers both performance and strong anonymity [24]. Snader and Borisov offer refinements to Tor's strategy, allowing an initiator to *tune* the performance (quantified in their work as bandwidth) of its anonymous paths [40] by defining the degree to which relay selection is biased in favor of bandwidth. At one extreme, initiators consistently choose relays with the highest bandwidth, achieving very high bandwidth paths at the expense of allowing a small subset of relays to view a significantly disproportionate amount of anonymous traffic [3, 26, 36]. At the other extreme, initiators may opt to favor anonymity while disregarding performance by selecting relays uniformly at random. Given the bandwidth requirements of the particular application, Snader's and Borisov's technique enables the sender to select a point in this anonymity-vs-performance spectrum. Similarly, we previously introduced tunable *link-based* routing [36], where initiators can weigh relay selection based on the expected e2e cost computed using link performance indicators such as latency, AS hop count, and jitter. We showed that biasing selection on link characteristics offers some anonymity benefits over node-based (i.e., bandwidth-weighted) techniques, since link-based routing reduces

"hotspot nodes" in the network that appear attractive to all initiators.

This is the first paper of which we are aware that addresses the related problem of *extensible anonymous routing*: given the variety of relay selection algorithms along with their individual performance and anonymity properties, how should applications produce anonymous paths that meet their specific needs? Rather than providing hardcoded relay selection policies, our proposed A^3 anonymity architecture allows applications to load routing policies at runtime. We show in Section 5 that existing anonymous routing techniques – including those described above – may be compactly represented in a few lines of declarative code. This allows applications to tune the degree of anonymity-vs-performance, and to select (and combine) different relay selection techniques.

This work extends our earlier proposal [38] in which we introduce the concept of using coordinate embedding systems [6, 5] – decentralized algorithms that efficiently map pairwise network distances into virtual coordinates – to produce anonymous paths with low latency. In contrast to our proposal, this paper presents a complete implementation of the A^3 anonymity system and describes techniques for combining multiple sources of information (including coordinate embedding systems) to form high performance paths. This paper also extends our recent work in which we introduce link-based routing [36]. In contrast to that work, the A^3 anonymity infrastructure proposed in this paper combines support for both node- and link-based relay selection strategies. Additionally, this work introduces the use of declarative networking [20] to both represent relay selection policies as well as instantiate anonymous paths.

3 Background on Declarative Networking

Given our use of *declarative networking* in A^3, we begin by providing some background. The high level goal of declarative networking is to enable the construction of extensible architectures that achieve a good balance of flexibility, performance, and safety. One specifies a declarative networking protocol as a set of queries in a high-level language. Because such a specification expresses *what* a program achieves as opposed to *how* it operates (the latter style is referred to as *imperative* programming), declarative queries are a natural and compact way to implement a variety of applications – especially routing protocols and overlay networks, which often may be expressed as a set of *recursive* queries. For example, path-vector and distance-vector routing protocols can be expressed in only a few lines of code [20], and the Chord [41] distributed hash table in 47 lines of code [19][1]. When compiled and executed, these declarative specifications perform efficiently relative to imperative implementations.

3.1 Datalog

Our A^3LOG declarative language is primarily based on Datalog [29]. A Datalog program consists of a set of possibly recursive declarative queries, also referred to as *rules*. Each rule has the form `q :- p1, p2, ..., pn.`, which can be read informally as "`p1` and `p2` and `...` and `pn` implies `q`". Here, `q` is the *head* of the rule and `p1, p2, ..., pn` is a list of *literals* that constitutes the *body* of the rule. Literals are either *predicates* (also called *relations*) with *attributes* (variables or constants) or boolean expressions that involve function symbols (including arithmetic) applied to attributes. A^3LOG extends Datalog by allowing the specification of rules with multiple head literals, i.e. rules of the form `q1, q2, ..., qm :- p1, p2, ..., pn.` A rule of this form is short-hand for the set of m rules where the ith rule is of the form `qi :- p1, p2, ..., pn.` A Datalog program is said to be recursive if a *cycle* exists through any predicate – such as when a predicate that appears once in a rule's body appears in the head of the same rule. A recursive Datalog program is continuously executed until a *fixpoint* is reached, i.e. no new facts are derived.

The order in which the rules are presented in a program is semantically immaterial; likewise, the order predicates appear in a rule is not semantically meaningful. Conventionally, the names of predicates, function symbols, and constants begin with a lowercase letter, while variable names begin with an uppercase letter. Function calls are prepended by `f_`. An aggregate construct, which defines an operation on multiple results from the rule body, is represented as a special function in the rule head with its attribute variables enclosed in angle brackets (`<>`). To support anonymous relay selection and path instantiation, A^3LOG enhances Datalog with *cryptographic functions*, *random* and *ranking aggregates*, and *composability*. We defer discussion of these additions to Sections 5- 6.

3.2 First Example: All Pairs Reachability

We illustrate A^3LOG using a simple example of two rules that compute all pairs of reachable nodes in a network.

```
r1 reachable(S,N) :- neighbor(S,N).
r2 reachable(@N,D) :- neighbor(S,N), reachable(S,D).
```

[1] In contrast, MIT's imperative implementation of Chord is several orders of magnitude larger.

Rules r1 and r2 specify a distributed transitive closure computation that derives all pairs of nodes that can reach each other through paths of neighbors. The rules take as input a local neighbor table stored at each node S (each fact in the neighbor(S,N) relation denotes that N is a neighbor of S). Rule r1 is a regular Datalog rule (i.e. executed locally at a node to derive local facts); it computes all pairs of nodes reachable within a single hop from all input neighbor links. Rule r2 expresses that "if N is the neighbor of S, and S can reach D, then N can reach D." The output of interest is the set of all reachable(S,D) facts, representing reachable pairs of nodes from S to D. By modifying this simple example, we can construct more complex routing protocols, such as the distance vector and path vector routing protocols.

Rule r2 introduces the *location specifier* – the argument prefixed with the @ symbol – which denotes the location of each fact derived by the rule head. In rule r2, all derived reachable(N,D) facts are exported based on the address encoded in their first attribute, (@N). This means that the execution of rule r2 results in each node propagating its reachability information to its neighbors until a distributed fixpoint (i.e. no new facts being derived) is reached.

A^{3}LOG queries are compiled and executed either locally – such as a regular Datalog rule like r1 – or in a distributed fashion, as in r2. In A^3, initiators specify relay selection policies using local rules (Section 5), and path instantiation using distributed rules (in fact, a series of distributed recursive queries as we show in Section 6).

A^{3}LOG shares a similar execution model with the Click modular router [18], which consists of elements that are connected together to implement a variety of network and flow control components. In the case of A^{3}LOG, these elements include database operators (such as joins, aggregation, selections, and projections) that are directly generated from queries. Reference [19] provides more details on the compilation process and execution model used in declarative networking that A^3 adopts.

3.3 Materialized Soft-state Tables and Events

Declarative networking incorporates a *soft-state* storage model, where each relation has an explicit "time to live" (TTL) or lifetime. All facts in the relation must be periodically updated before their TTL expires, or they are deleted.

A^{3}LOG supports soft-state through the materialize [19] directive, which specifies the TTL of each relation. A materialize directive has the form: materialize(Relation, Timeout, Max_entries, Keys), where Relation is the name of the relation, Timeout is the maximum time in seconds that any fact in the relation may persist, Max_entries is the maximum facts allowed in a relation before facts are ejected according to a FIFO policy, and Keys specifies the relation's primary keys. If a fact is derived with the same primary keys as an existing fact in the same relation, the new fact *replaces* the old one, and the TTL is restored.

If a relation has no corresponding materialize directive, it is treated as an *event* predicate with zero lifetime. Event predicates – whose names are prefixed with an "e" – are used to denote transient tables used as input to rules.

4 A^3 Design Goals and Architecture

A^3 is a flexible and extensible anonymity system in which protocol designers publish their particular relay selection and path instantiation algorithms along with a description of their corresponding performance and anonymity tradeoffs. In contrast to existing anonymity systems in which an immutable relay selection algorithm is hardcoded into the anonymity service, A^3 allows the sender to provide a *relay selection policy* that precisely specifies the manner in which relays are chosen for its anonymous paths. The A^{3}LOG policy language (Section 5) enables the application to not only intelligently tune relay selection in favor of performance or anonymity [40, 36], it also allows the application to easily define its individual characterization of performance in terms of bandwidth, latency, loss, jitter, etc., or some combination of the above. In addition to supporting flexible relay selection, A^3 also permits the customization of *path instantiation policies* (Section 6).

A^3's use of declarative networking provides the capability for applications to rapidly customize and refine the policies that best meet their application constraints. However, our system does not preclude similar tuning outside the use of declarative languages. A user of A^3 who is not familiar with declarative policy languages can, for example, simply download and install an A^{3}LOG policy that produces low latency and low jitter paths for his VoIP application, while using a different policy to deliver high downstream throughput for anonymous web browsing.

Since the anonymity offered by an anonymous path depends in no small part on the mechanisms for relay selection [24, 36, 26] and path instantiation, policies should be used with extreme caution until their security properties can be fully understood. A thorough review of the performance and anonymity properties of various

relay selection and path instantiation algorithms is outside the scope of this paper. Our goal in this paper is to provide a flexible architecture for developing, testing, and studying path strategies and implementations (though A^3 constitutes a very useful tool for conducting security evaluations).

System Overview. An application, or a proxy acting on the application's behalf, provides relay and path instantiation policies that reflect the application's communication requirements. Figure 1 shows the architecture of the A^3 client running on the initiator's host. The *Relay Selection Engine* interprets the initiator's relay selection policy and applies that policy to produce (but not instantiate) an anonymous path consisting of relays from the *Local Directory Cache*. To populate the cache, the A^3 instance periodically contacts a *Directory Server* to ascertain membership information – that is, a listing of available relays – and, optionally, one or more *Information Providers*. Information Providers are data aggregating services that report performance characteristics of relays (e.g., bandwidth) and links (e.g., the latency between two relays). The Relay Selection Engine uses cached data to generate paths that conform to the provided relay selection policy.

Once the Relay Selection Engine produces a path, the *Forwarding Engine* instantiates that path according to the provided path instantiation policy. After path establishment, a *Proxy Service* on the local machine intercepts the application's traffic and relays it through the anonymous path. Likewise, incoming data from the anonymous channel is transparently forwarded through the Proxy Service to the application. To forward upstream and downstream packets, each relay also includes a Forwarding Engine.

Below, we describe each component of A^3 in more detail.

4.1 Information Providers

To support non-trivial relay selection policies, A^3 makes use of *Information Providers* (also referred to as *Providers*) that aggregate node and/or link performance data. Policies may utilize such information to more precisely define their requirements (e.g., "include only relays that have been online for at least an hour").

A^3 imposes few restrictions on the types of Information Providers. Each Information Provider is interfaced through an *adapter* that resides on the A^3 relay. Adapters are small programs or scripts that periodically query a Provider for new information, storing the results in the Local Directory Cache. Our current implementation includes adapters for the Vivaldi [6] embedded coordinate system (described below) and CoMon [27], although others can be easily constructed.

Network Coordinate Information Providers Traditional anonymous relay selection algorithms (most notably Tor [9], and the refinement proposed by Snader and Borisov [40]) bias selection in favor of relays that advertise high bandwidths. However, in addition to bandwidth, an application may also prefer paths that exhibit low latency. Unlike bandwidth, latency is not a *node characteristic* that can be associated with an individual relay. Rather, latency is a *link characteristic* that has meaning only when defined in terms of a connection between a pair of relays.

Given that there are $\binom{N}{2}$ links in a network composed of N relays, maintaining link characteristics for all relays in the anonymity network is infeasible. One practical solution to succinctly capture pairwise link latencies is via the use of *virtual coordinate embedding systems* (also called network coordinate systems). These distributed algorithms enable the pairwise latencies between all participating relays to be estimated to high accuracy with low overhead. Network coordinate systems, such as Vivaldi [6], PIC [5], NPS [25], and Big Bang Simulation [34] map each relay to multidimensional coordinates such that the Euclidean distance between any two relays' coordinates corresponds to the latency between the pair. By representing pairwise distances using N virtual coordinates, these systems effectively linearize the information that must be stored and maintained by the Information Provider.

Coordinate systems use distributed algorithms in which each participant periodically measures the distance between itself and a randomly selected peer. By comparing the empirical measurement with the Euclidean distance between the two nodes' coordinates, the relay can adjust its coordinate either towards (in the case of over-estimation) or away from (for under-estimation) the neighbor's coordinate. Although network distances cannot be perfectly represented in Euclidean space due to the existence of triangle inequality violations on the Internet, virtual coordinate systems efficiently estimate pairwise distances with very low error [6]. Since network coordinate systems require only periodic measurements (on the order of a single ping every 15 seconds), participation in the system does not incur a significant bandwidth cost.

A *Network Coordinate Information Provider* maintains the current coordinates of the relays in the A^3 net-

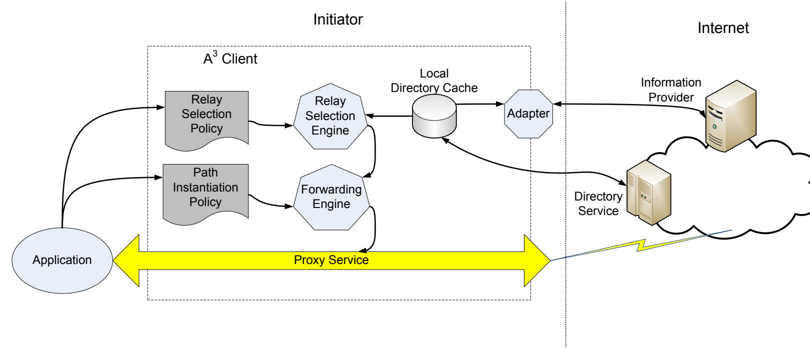

Figure 1. The A^3 architecture.

work. Relays periodically send updates to the Provider whenever its coordinate changes from its last reported value (e.g., by more than 10ms).

Unfortunately, the distributed nature of coordinate systems make them particularly vulnerable to insider manipulation. Recent studies [16] on Vivaldi have shown that when 30% of nodes lie about their coordinates, Vivaldi's accuracy decreases by a factor of five. Attacking the coordinate system provides a vector for an adversary to either prevent high performance routing or bias routing decisions in favor of relays under their control. Fortunately, practical coordinate protection techniques may be applied on top of the embedding system to protect the veracity of advertised coordinates [5, 33, 15, 43, 37]. Often, these coordinate security techniques rely on spatial and temporal heuristics to spot false coordinate advertisements [43], utilize a small set of trusted surveyor nodes [33, 15], or assess coordinate accuracy using a distributed voting protocol [37]. Network Coordinate Information Providers should employ such services to ensure that the coordinate information it provides to A^3 nodes is trustworthy.

Relay-Assisted Information Providers Relays have access to a significant number of local performance indicators. For instance, a relay can measure its current upstream and downstream throughput, processor usage, and available memory, and estimate its bandwidth capacity. Such information can be collected and stored in a *Relay-Assisted Information Provider*. The CoMon Monitoring Infrastructure [27] that operates on the PlanetLab testbed [28] is one such example.

As has been pointed out by Øverlier [26] and others [40, 3, 36], malicious relays may purposefully attract a large fraction of anonymous traffic by falsely advertising favorable performance, consequently increasing their view of traffic in the anonymous network. To mitigate such attacks, Snader and Borisov propose the use of *opportunistic measurements* in which relays report the observed throughput of their network peers. The Provider (or in their case, the directory service) reports the median of the reported measurements [40]. Similar protection schemes are applicable to A^3 Information Providers. Here, relays report the bandwidth and responsiveness of peer relays with whom they interact. Certain metrics (e.g., memory usage) cannot easily be probed by remote parties, and if reported by Information Providers, should be treated with some degree of skepticism by the relays that make use of them.

Other Potential Information Providers There have been a number of proposals (e.g. iPlane [21], IDMaps [11], OASIS [12], and Meridian [42]) that attempt to succinctly map the structure of the Internet and provide estimates of latency and (in some cases) bandwidth between arbitrary Internet hosts. Such systems have typically been deployed to provide proximity-based routing [38], neighbor selection in overlays [7], network-aware overlays, and replica placement in content-distribution networks. However, these systems are also applicable to anonymity services in which the initiator is interested in discovering the cost of routing through a particular relay. By constructing adapters that access the interfaces to these systems, initiators can construct relay selection policies that take advantage of such services. The mechanism for inserting polled information into the Local Directory Cache is described in the following section.

4.2 Other Components of A^3

We briefly describe the other components of the A^3 system, namely the directory service, local directory cache, relay selection engine, and forwarding engine.

4.2.1 Directory Service

Node discovery is facilitated by a *Directory Service* (or simply Directory) that maintains membership information on all the relay nodes currently participating in the A^3 platform. Relays that join the A^3 network publish their network address and public key to the Directory Service. Initiators periodically poll the Directory to discover peer nodes that may potentially be used as routers in anonymous paths.

The Directory Server represents a central point of failure in the network. If the server becomes unavailable, then initiators cannot discern the network addresses of available relays. If the Directory is malicious or becomes compromised, then it can answer queries with only the identities of misbehaving relays.

Tor reduces this risk by establishing multiple semi-trusted *directory authorities* [1]. The authorities periodically vote on a summary of the network (that is, the relays that constitute the network), and disseminate signed *consensus documents* to Tor routers. The routers further redistribute this network information to other routers and to clients. Clients verify signatures on consensus documents based on certificates shipped with the Tor binary. Although our current implementation uses a single Directory Service, such protections can be straightforwardly applied to A^3.

4.2.2 Local Directory Cache

The *Local Directory Cache* periodically queries and stores performance data from Information Providers. The rate at which the cache polls Providers affects both the freshness of cached data as well as the relay's communication overhead. The tradeoff between update intervals and bandwidth costs depends on the rate at which performance characteristics change in the network, and is explored in more detail in Section 7.

The Local Directory Cache uses adapters to query the various Information Providers, storing the results in tables that are accessible by the Relay Selection Engine. Adapters define tables using the `materialize` keyword as described in Section 3. For example, given the following A^3LOG statements,

```
materialize(tBandwidth, Infinity, Infinity, keys(1)).
materialize(tVivaldiCoordinates, Infinity,
            Infinity,keys(1)).
tBandwidth("10.0.0.1", 1000, 500, 3000).
tVivaldiCoordinates("10.0.0.1", [10,-6]).
```

the Local Directory Cache will create two tables: `tBandwidth` and `tVivaldiCoordinates`. The former holds the address of a remote node, its upstream bandwidth, downstream bandwidth, and bandwidth capacity. The `keys(1)` argument specifies that an existing tuple should be replaced if a new tuple arrives with the same first field (the network address). Similarly, the latter table stores the coordinates of a remote node. The two example `tBandwidth` and `tVivaldiCoordinates` statements insert data into the respective table. Such statements are executed by the adapter as new data is polled from Information Providers.

4.2.3 Relay Selection Engine

The Relay Selection Engine provides the flexibility that enables applications to communicate their routing requirements. At runtime, applications provide relay selection policies that specify their individual routing criteria. Using the information stored in the Local Directory Cache, the Relay Selection Engine forms routes according to the specified policy. The participants of generated paths are relayed to the Forwarding Engine (see below) that instantiates the path. Relay selection is explored in more detail in Section 5.

4.2.4 Forwarding Engine

The *Forwarding Engine* consists of a declarative networking engine enhanced with low-level cryptographic primitives. The Forwarding Engine provides methods for composing these primitives to form high-level operations. For example, the one-way authentication and symmetric key-exchange primitives used in Tor path instantiation are constructed by composing RSA digital signatures with Diffie-Hellman key exchange.

The Forwarding Engine instantiates the anonymous path provided by the Relay Selection Engine according to rules specified in the *path instantiation policy*. Additionally, the Forwarding Engine supports message relay over instantiated anonymous paths. That is, the Forwarding Engine is used both to construct paths as well as to relay application messages over the anonymous route. In the case of Onion Routing, Tor, and Crowds, the rules for path instantiation and message relay are distributed recursive queries. We revisit the routing engine in Section 6.

5 Relay Selection Policies

In this section, we demonstrate how a variety of strategies used by the relay selection engine can be expressed using the declarative framework. These rules are then executed by a declarative networking engine to implement the selection policies. In presenting the relay

Algorithm	Reference	Description	Benefits	Example Usage
RANDOM	Section 5.1	Relays selected uniformly at random	Produces low node prevalences	Email mixing
TOR [9]	Section 5.1.1	Relays biased proportionally to bandwidth	High bandwidth and network utilization [24]	Web browsing
SNADER-BORISOV [40]	Section 5.1.2	Tunable bias towards bandwidth	Tunable anonymity and performance	File transfer
CONSTRAINT	Section 5.2.1	Specification of end-to-end performance requirements	Expresses communication requirements	VoIP
WEIGHTED [36, 35]	Appendix A	Bias relay selection in favor of link-properties	Extends support to multiple metrics (latency, jitter, etc.)	Streaming multicast
HYBRID	Section 5.2.3	Combines above techniques	Supports highly flexible routing policies	Video conferencing

Table 1. Example relay selection policies

selection examples in A^{3}LOG, we also highlight extensions we have made to canonical declarative languages, including customizable *random aggregates*, *ranking aggregates*, and a *tuple list type* convenient for selecting and manipulating relays and coordinates.

Our goal in this section is to highlight the flexibility, ease of programming, and ease of reuse afforded by a declarative query language. We show that routing protocols can be expressed in a few A^{3}LOG rules, and additional protocols can be created by simple modifications to previous examples. We present examples of well-known node-based and link-based strategies, as well as hybrid strategies that select relays based on a combination of node and link metrics. In addition to the specific rules shown, users of A^3 can also maintain several sets of rules, as well as dynamically specify rule parameters in new path requests. This allows for extremely flexible, on-the-fly path generation tuned for specified parameters without additional modification to A^{3}LOG code.

In all our examples below, we assume that each A^3 node has a background process that periodically polls the Directory Server to gather candidate nodes. At each node, the returned information is stored in a `node` table, indexed by the IP address of the node. Additional attributes obtained from Information Providers are included in this table (for example, the network coordinate of the node, its bandwidth measurements, etc.)[2]. These node and measurement data are then used as input to A^3 for executing A^{3}LOG rules that will select candidate relays. In all our example programs, the output of interest is an `ePathResult(Src,Dst,P)` tuple, where `P` is the list of relay node tuples (which includes their addresses, bandwidth, coordinates or any other attributes relevant to the query) from `Src` to `Dst`. The properties of the example relay selection policies are summarized in Table 1.

[2]Note that these attributes correspond to the attributes in `tBandwidth` and `tVivaldiCoordinates` tables described in Section 4.2.2. For ease of exposition, we refer to them as attributes in the `node` materialized table.

5.1 Node-based Relay Selection

As its name suggests, node-based relay selection primarily selects nodes based on node characteristics, typically bandwidth. The following rule `r1` shows an example A^{3}LOG program being executed by an A^3 node. Given a request to generate a path from `Src` to `Dst`, the program randomly selects three relay nodes other (excluding `Src` and `Dst`).

```
r1 ePathResult(Src, Dst, RAND(3)<IP>) :-
     ePathRequest(Src, Dst), node(IP),
     Src != IP, Dst != IP.
```

Rule `r1` takes as input a path request, in the form of an event tuple `ePathRequest(Src,Dst)`, where `Src` is typically the address of the node that issued the request, and `Dst` is the address of the responder. Rule `r1` is essentially a typical database query with group-by attributes (`IP` in this case) and a *random aggregate*.

Unlike a regular aggregate that computes, for instance, the minimum and maximum value, a random aggregate is a function of the form $\texttt{RANDAGG}(a_1, a_2, ..., a_m)\langle p_1, p_2, ..., p_n\rangle$ that takes in $(a_1, a_2, ..., a_m)$ as m argument parameters, and n arguments in $\langle p_1, p_2, ..., p_n\rangle$ that denote the output (projection) attributes of the resulting group-by value. Given result tuples generated in the rule body, for each group-by value, `RANDAGG` performs the appropriate random selection algorithm based on its function definition, and then returns a list of tuples with the appropriate n attributes being projected from the results.

For instance, in rule `r1`, `RAND(3)<IP>` is a random aggregate with argument 3 and projecting by IP. With these parameters, the aggregate will return 3 randomly selected nodes without replacement from the result of executing the rule body. The output of executing the rule `r1` is the `ePathResults(Src,Dst,P)` event tuple, where `P` is a list of tuples each containing the IP address field of the selected relay nodes from `Src` to `Dst`. The additional selection predicates in `r1` ensure that neither `Src` or `Dst` are selected as relay nodes.

5.1.1 Bandwidth-weighted Selection

The above selection strategy randomly chooses three nodes as relays without taking into consideration their node characteristics. As an enhancement, the following rules implement Tor's relay selection [8] and selects nodes with probability weighted by their bandwidth. A node with higher bandwidth has a greater probability of being selected, and the likelihood of selection relative to other nodes is linearly proportional to bandwidth.

```
t1 eCandidateRelay(Src, Dst, PathsSoFar,
                   RANDWEIGHTED(1,BW)<IP>) :-
     ePathRequest(Src, Dst, PathsSoFar),
     node(IP,BW), Src != IP, Dst != IP.

t2 ePartialPath(Src, Dst, PathNew) :-
     eCandidateRelay(Src,Dst,PathsSoFar,Relay),
     PathsSoFar.inPath(Relay[0]) = false,
     PathNew = f_append(PathsSoFar, Relay[0]).

t3 ePartialPath(Src, Dst, PathsSoFar) :-
     eCandidateRelay(Src,Dst,PathsSoFar,Relay),
     PathsSoFar.inPath(Relay[0]) = true.

t4 ePathRequest(Src, Dst, P) :-
     ePartialPath(Src, Dst, P), f_size(P) < 3.

t5 ePathResult(Src, Dst, P) :-
     ePartialPath(Src, Dst, P), f_size(P) = 3.
```

The initial route request is triggered by the requesting event `ePathRequest(Src,Dst,())`, where the last attribute is the current path initially initialized to the empty list `()`. Rule `t1` is similar to the earlier rule `r1`, except that it uses the aggregate function `RANDWEIGHTED(1,BW)<IP>` which selects one tuple randomly from the tuples derived from executing the rule body, with probability linearly weighted by the bandwidth attribute `BW`. (To bias the selection using another metric such as average node load, one would simply have to modify the parameter to `RANDWEIGHTED`.) The resulting output is a list containing one tuple, which can be retrieved as the first element of the list (indicated by the index `[0]`) followed by a projection on the `IP` field.

Rules `t2` and `t3` generate a new `ePartialPath` if the chosen `Relay` is not already in the current partial path; otherwise, they add the relay's IP to the path. The process repeats in `t4` if the number of relays selected is less than three. Eventually, the resulting path `ePathResult` is returned via rule `t5` when three relay nodes have been chosen.

5.1.2 Tunable Performance/Anonymity Selection

Snader's and Borisov's recent proposal [40] introduces a tunable weighting system that allows the initiator to trade between anonymity and performance. Briefly, their proposal defines the family of functions

$$f_s(x) = \begin{cases} \frac{1-2^{sx}}{1-2^s} & \text{if } s \neq 0 \\ x & \text{if } s = 0 \end{cases} \quad (1)$$

where s is a parameter chosen by the initiator that allows for a tradeoff between anonymity and performance. After ranking the relays by bandwidth, the initiator chooses the relay with index $\lfloor n \cdot f_s(x) \rfloor$, where x is chosen uniformly at random from $[0,1)$, and n is the number of nodes. By applying higher values of s, the initiator is able to more heavily bias her selections towards bandwidth. On the other hand, for $s = 0$, a relay is chosen uniformly at random [40]. Each relay is selected independently and without replacement according to the distribution imposed by Eq. 1.

Snader and Borisov's algorithm may be represented in A^{3}LOG by modifying the `t1` rule from above into two rules:

```
s1 eRelayList(Src,Dst,PathsSoFar,S,SORT(BW)<IP>) :-
     ePathRequest(Src,Dst,PathsSoFar,S), node(IP,BW),
     Src != IP, Dst != IP.

s2 eCandidateRelay(Src,Dst,PathsSoFar,Relay) :-
     eRelayList(Src,Dest,PathsSoFar,S,SortedRelayList),
     sbRand = (1 - 2^(S * f_rand01())) / (1 - 2^S)),
     Relay = f_selectIndex(SortedRelayList,sbRand).
```

`SORT(BW)<IP>` is a *ranking aggregate* which follows a similar syntax as the random aggregates. It takes all the resulting tuples derived from executing the rule body, performs a sort using the `BW` attribute, and then returns the projected field `IP` as a nested tuple based on the sort order. Hence, the `SortedRelayList` attribute of `eRelayList` will include a sorted list of `IP` tuples. Rule `s2` applies Eq. 1 to generate a biased random variable which is then used to index into the list and select a relay.

5.2 Link-based Selection

The previous examples have focused exclusively on *node characteristics* – performance metrics (i.e., bandwidth) that may be attributed to individual relays. In *link-based path selection* [36], the e2e performance of a path is computed by aggregating the cost of all links that comprise the path, where cost is defined in terms of *link characteristics* such as latency, loss, and jitter. (While bandwidth is a node-based characteristic, it can also be represented as a link characteristic by considering the measured available bandwidth on a link connecting two nodes.) The use of link rather than node characteristics enables not only more flexible routing (since initiators can construct anonymous routes that meet more specific communication requirements), but also offers better protection of the identities of the communicating parties [36].

In these examples, the table of node information gathered from the directory service and Information Providers is stored in the format `node(IP, Coord)` and includes nodes' network addresses and virtual coordinates.

5.2.1 End-to-end Constraint-Based Selection

The simplest form of link-based selection is based on selecting paths that meet e2e constraints. Rules `c1-c4` result in the selection of three relay nodes where the e2e latency is less than `Limit`.

```
c1 eCandidatePath(Src,Dst,Limit,
   RAND(3)<IP,Coord>) :-
     ePathRequest(Src, Dst, Limit),
     node(IP, Coord), Src!=IP, Dst!=IP.

c2 ePathCost(Src, Dst, Limit, P, Cost) :-
     eCandidatePath(Src, Dst, Limit, P),
     Cost =
       f_coorddist(Src.Coord, P[0].Coord) +
       f_coorddist(P[0].Coord, P[1].Coord) +
       f_coorddist(P[1].Coord, P[2].Coord) +
       f_coorddist(P[2].Coord, Dst.Coord).

c3 ePathRequest(Src, Dst, Path) :-
     ePathCost(Src, Dst, Limit, Path, Cost),
     Cost > Limit.

c4 ePathResult(Src, Dst, Path) :-
     ePathCost(Src, Dst, Limit, Path, Cost),
     Cost <= Limit.
```

Rule `c1` is similar to the earlier random selection rules that select a random set of three relay nodes. Here, however, the `Coord` field is also projected for use in rule `c2`. Based on the three selected relays, `c2` computes the e2e path cost as the sum of the Euclidean distances of the coordinates. The process repeats (rule `c3`) until a path whose overall cost is less than `Limit` (an input variable) is selected (rule `c4`).

5.2.2 Tunable Performance/Anonymity Selection

In Appendix A, we additionally present a link-based path selection algorithm, WEIGHTED [36] that provides tunable performance and anonymity. The algorithm consists of two phases. In the first phase, the initiator rapidly generates (but does not instantiate) candidate paths consisting of three relays chosen uniformly at random without replacement. The initiator computes the e2e cost of each generated candidate path. In the second phase, the initiator sorts the candidate paths by their cost estimates, and then applies Eq. 1 to select the path.

5.2.3 Hybrid Selection

Although the above rules use a single metric when selecting a path, it is easy to combine multiple factors for relay selection.

The following rule selects a path whose minimum bandwidth is above `Thres` by a conditional join on nodes in the table, and then uses the coordinate embedding system to select a path with e2e latency less than `Limit`. In this example, the table of all nodes `node(IP, Coord, BW)` stores the virtual latency coordinate as well as bandwidth for each node IP. Interestingly, we need only make one change to replace `c1` (from 5.2.1) with `h1`:

```
h1 eCandidatePath(Src, Dst, Limit,
                 RAND(3)<IP,Coord,BW>) :-
     ePathRequest(Src, Dst, Limit, Thres),
     node(IP,Coord,BW),
     Src!=IP, Dst!=IP, BW>THRES.
```

Other hybrid policies using multiple metrics can be similarly constructed (for example, using the WEIGHTED policy only on nodes whose bandwidth is above a threshold).

6 Path Instantiation Policies

A^3's forwarding engine performs *path instantiation*, a process that establishes necessary network state at each selected relay to enable bidirectional data flow over an anonymous circuit between a given initiator and any destination. Unlike relay selection, which happens locally at the initiator, path instantiation is an inherently distributed operation, and thus exercises the distributed execution features of A^3LOG.

In this section, we demonstrate the use of A^3LOG for path instantiation. Due to space constraints, we primarily emphasize Onion Routing [30] and leave a discussion of Tor [9] and Crowds [31] to Appendix B. We evaluate the performance of our Onion Routing path instantiation implementation in Section 7.2.1.

We begin with a brief overview of the path instantiation scheme used by Onion Routing. After selecting a path consisting of one or more relays – called *onion routers* – the initiator sends a recursively encrypted message called an *onion* to the first hop of the selected path. Each *layer* of the onion contains the address of the next desired hop in the path, and seed material to generate symmetric keys shared with the initiator [3]. Public key cryptography ensures that every node can interpret exactly one layer of the onion. Each node removes its layer, generates keys from the seed material, and – if it is not the endpoint – forwards the remainder of the onion on to the next hop. The endpoint sends a confirmation message to the initiator backward along the newly-instantiated path.

[3] In practice, each layer also contains information about which cryptographic algorithm to use in each direction of the circuit, a timestamp, and a version identifier.

More precisely, if the relays in the anonymous path are $R_1, ..., R_n$ and $M_1, ..., M_n$ are the relays' corresponding onion layers, then the onion is encrypted as $E_{R_1}(M_1, E_{R_2}(M_2, ..., E_{R_N}(M_N)))$, where $E_X(W)$ denotes the encryption of message W using the public key belonging to X. In practice, only the key seed material is encrypted with the public key. The remaining data is encrypted using a symmetric key derived from the key seed material.

Onion routing specifies an additional link-layer protocol that governs how messages are exchanged between onion routers. For a discussion of this protocol, see [30].

6.1 Onion Routing in A^{3}LOG

Our A^{3}LOG implementation of Onion Routing requires 12 rules to specify path instantiation. These rules consist of three recursive computations: building the onion, relaying the onion along the path to establish state at each node, and forwarding a confirmation back along the path. We extended our implementation to support forwarding data along an instantiated path at a cost of five additional rules.

We briefly summarize the *schema* (data format) of the facts computed in relations at each node. All facts are indexed by a locally unique `CID` (circuit identifier). An initiator stores a `circuitPath(CID,Path)` fact that associates a circuit with a path representing the chosen relay nodes. The `Path` variable represents the result of the relay selection phase and is populated based on the `ePathResult` tuple. In addition, the initiator stores the current state of the circuit in the `circuitStatus(CID,Status)` relation. The value for `Status` may be either `BUILDING` or `ESTABLISHED`. As the path is being instantiated, the initiator and each intermediate relay creates a link-local identifier (`ACI`) for the circuit, stored along with the circuit's next relay in a `circuitForward(CID,ACI,Node)` fact. Similarly, the final relay and each intermediate relay stores the `ACI` generated by the previous `Node` in the `circuitReverse(CID,ACI,Node)` relation. At each relay, symmetric encryption keys (shared with the initiator) for forward and reverse cryptographic operations are stored in the `circuitKeys(CID,ForwardKey,ReverseKey)` relation. For each relay node, the initiator maintains these keys in the `circuitInitiator(CID, Relay, ForwardKeys, ReverseKeys)` relation.

It is worth noting that many of the relations used by the Onion Routing rules can also be used by Tor and Crowds. For example, all of these systems involve multiplexing traffic from multiple anonymous circuits over a single link, necessitating the use of per-circuit link-local unique identifiers. Also, in each system, paths are bidirectional, requiring intermediate nodes to store the next node in each of the forward and backward directions. To differentiate between different types of paths, we use a `circuitType(CID, Type)` relation.

Below, we highlight the use of A^{3}LOG via the following three rules (`oc1-oc3`) that express the local recursive computation of generating an Onion at the initiator:

```
oc1 circuitPath(CID, Path),
    circuitStatus(CID, "BUILDING"),
    circuitForward(CID, ACIForward, FirstRelay),
    eCreateOnion(CID,LastRelay,
                 RemainingPath,FirstLayer) :-
       ePathResult(_,_,Path),
       FirstRelay=f_first(Path).IP,
       LastRelay=f_last(Path).IP,
       ACIForward=f_gen_aci(),
       RemainingPath=f_removeLast(Path),
       CID=f_gen_cid(), FirstLayer={}.
```

Rule `oc1` is triggered upon insertion of a new path. It generates state at the initiator for the new circuit, including the local `CID` and link-local `ACI`. These are respectively used to differentiate between circuits at a given node and circuits on a given link. In addition, `oc1` associates the new circuit with its path representation, and a status describing the current stage of the circuit's lifecycle – `BUILDING` in this case indicates that the circuit is currently being instantiated and is not yet ready for use. Rule `oc1` triggers the recursive rule, `oc2`, through the `eCreateOnion` event:

```
oc2 eCreateOnion(CID,NextRelay,
                 RemainingPath,NextLayer) :-
       eCreateOnion(CID, CurrentRelay,Path, PrevLayer),
       f_size(RemainingPath) != 0,
       NextRelay = f_last(Path).IP,
       RemainingPath = f_removeLast(Path),
       encryptOnion(CID, CurrentRelay,
                    PrevLayer, &EncryptedLayer),
       NextLayer={NextRelay, EncryptedLayer}
```

The `eCreateOnion` event represents an intermediate step of circuit instantiation. Its first argument references the `CID` of the circuit being created, its second notes the most recently added relay, and its third contains the intermediate representation of the onion. Note that onions are built outwards from the innermost layer. We denote the innermost layer as an empty list, as this layer will be interpreted by the ultimate relay in the circuit, who does not extend the path any further.

Rule `oc2` calls the `encryptOnion` *Composable View* (CView, described in Section 6.2), which encrypts the previous layer of the onion. Rule `oc2` is linearly recursive and will continue to trigger itself and derive new facts as long as `RemainingPath` is nonempty. Each invocation of the rule removes a relay

node from RemainingPath as it adds a layer of encryption. Upon reaching the terminating condition – when RemainingPath is empty – rule oc3 is triggered:

```
oc3 eOnionMessage(@FirstRelay, ACI, CompleteOnion) :-
       eCreateOnion(CID, CurrentRelay,
                    RemainingPath, PrevLayer),
       f_size(RemainingPath) = 0,
       circuitForward(CID, ACI, FirstRelay).
       encryptOnion(CID, CurrentRelay,
                    PrevLayer, &CompleteOnion).
```

Rule oc3 calls encryptOnion to encrypt the final layer of the onion, and sends the completed onion to the first relay node (via the location specifier @FirstRelay) read from the circuitForward relation. Upon receiving the onion, each intermediate relay will peel off and decrypt a layer of the onion (using the decryptOnion CView), extract the location of the subsequent hop, and recursively forward the onion. We omit these rules due to space constraints.

The observant reader will note that the above rules do not implement Onion Routing's link-layer protocol. One may easily specify this protocol in A^{3}LOG by adding a layer of indirection to any rule that sends a high-level anonymous message. We omit the specification here, as it involves relatively mundane serialization, encapsulation, and encryption.

6.2 Composable Virtual View for Onion Encryption

In order maximize reusability between different path instantiation protocols and enable re-configurable encryption, we leverage *Composable Virtual Views* (CViews) [22] to express high-level cryptographic primitives. A CView is a user-defined function implemented in A^{3}LOG. A call to a CView may only occur in the body of a rule, and has the following syntax:

```
viewName(K1,K2,...,Kn, &R1,&R2,...,&Rm)
```

Each CView has a set of input attributes – shown above as K1,K2,...Kn – which must be bound at the beginning of the call to the CView, and a set of return attributes, &R1,&R2,...,&Rm, that are returned by the call. Note that CViews do not augment the expressive power of the A^{3}LOG language but rather provide modularity. In fact, any rule that uses CViews can be rewritten as a series of regular A^{3}LOG rules using a rewrite [22].

We illustrate the encryptOnion CView used by the above rule:

```
def encryptOnion(CID, Node, Data_in, &Data_E) {
   eo1 circuitInitiatorKeys(CID, Node,
                            Key_forw, Key_back)
       this.return(Data_E) :-
          this.init(CID, Node, Data_in),
          KeySeed = f_genKeySeed(),
          Key_onion = f_sha1(KeySeed),
          Key_forw = f_sha1(Key_Onion),
          Key_back = f_sha1(Key_forw),
          Payload_E = f_symEncrypt(Key_onion,Data_in),
          publickey(Node,PubKey),
          KeySeed_E = f_asymEncrypt(PubKey,KeySeed),
          Data_E=[KeySeed_E, Payload_E].
}
```

The built-in predicates this.init and this.return respectively specify the input values and return values to/from the CView. Rule eo1 generates key seed material and iteratively applies the SHA-1 hash function to derive three symmetric keys to be shared between the initiator and a given Node in the circuit: (i) Key_onion, used for encrypting the layer of the onion (except for KeySeed) destined for Node; (ii) Key_forw, used for cryptographic operations on data sent forward from the initiator; and (iii) Key_back, used for cryptographic operations on data sent backward to the initiator. These keys are stored at the initiator in the circuitInitiatorKeys relation[4]. Rule eo0 then employs Key_onion to encrypt Data_in, which consists of the next node in the circuit, and the previous layer of the onion. The KeySeed is then encrypted with the public key of Node.

Because CViews can clearly separate the cryptographic operations from the specification of the protocol, one can easily tune the encryption by customizing the above encryptOnion (and the corresponding decryptOnion) CView. Furthermore, CViews can facilitate reusability of these high-level cryptographic primitives across different path instantiation protocols.

7 Evaluation

In this section, we describe our implementation of A^3 and present measurement studies that demonstrate its ability to produce flexible anonymous paths at low performance overhead. To examine the scalability aspects of our implementation, we evaluate A^3 both in simulation using real-world network traces as well as an actual deployment on the PlanetLab testbed.

7.1 Implementation and Experimental Setup

A^3's Relay Selection and Forwarding Engines has been implemented as an extension to the P2 [19] declarative networking system, with enhancements to enable the various new features of A^{3}LOG as described in Sections 5 and 6. All relay selection policies used our experiments produce paths with three anonymizing relays.

[4] Note that circuit_initiator does not contain Key_onion – this key is only used for cryptographic operations on the onion sent for path instantiation, and need not be persisted.

We utilize a MySQL database server as our Directory Service. A^3 nodes register with the service and query for membership by sending SQL queries and updates to the server. Membership information that has not been refreshed within 15 minutes is purged from the directory. Although the MySQL protocol adds a small degree of unnecessary bandwidth overhead, directory fetches and updates occur infrequently. To minimize such overhead, compression is enabled in both A^3 clients and the database server. The Local Directory Cache, which queries the Directory Service for membership information, stores the results in `materialize` tables that are accessible to the Relay Selection Engine.

The A^3 system executes in simulation mode by taking as input a network trace consisting of pairwise latency measurements [17] and running several A^3 instances on a single machine. Actual implementation is achieved by running each A^3 instance on PlanetLab, interfacing with the Directory Service and Information Providers.

Our experiments utilize two Information Providers. Each relay (whether running in simulation mode or on PlanetLab) participates in a Vivaldi [6] virtual coordinate embedding system. Relays update the Network Coordinate Information Provider (here, the MySQL database used as our Directory Service) with its new coordinate when the Euclidean distance between coordinate updates exceed 10ms. Additionally, our PlanetLab experiments utilize the CoMon PlanetLab Monitoring Infrastructure [27] to retrieve peers' bandwidth, memory, and CPU usage information.

7.2 Relay Selection Evaluation

A^3 is a flexible anonymity framework that enables the rapid development and deployment of relay selection algorithms. To demonstrate A^3's ability to parse a diverse set of relay selection policies, we evaluate the performance characteristics of generated paths and compare them to their expected values.

7.2.1 Simulation-based Evaluation

Figure 2 shows the cumulative distribution of end-to-end (e2e) latencies using our A^3 implementation in simulation mode with the WEIGHTED relay selection policy [36]. The x-axis of the graph plots the achieved e2e latency, while the y-axis indicates the fraction of paths that has at most that latency. As input to the trace-driven simulator, we utilize the King dataset [17], a collection of pairwise latencies collected from the Internet using the King method [14]. As can be seen from the figure, WEIGHTED reduces the e2e RTT of anonymous paths when compared to RANDOM. For example, the median RTT of anonymous paths decreases 30% from 261ms to 184ms when WEIGHTED is used. Similarly, Figure 3 shows the achieved performance, measured as the e2e bandwidth of anonymous paths, using various node-based relay selection policies. Paths were constructed using bandwidth information retrieved from Tor directory servers. As expected, the Tor routing policy produces paths with significantly greater bandwidths than random selection. The Snader-Borisov algorithm achieves tunable performance results – as the value of s increases, the effective e2e bandwidth of anonymous paths also increases.

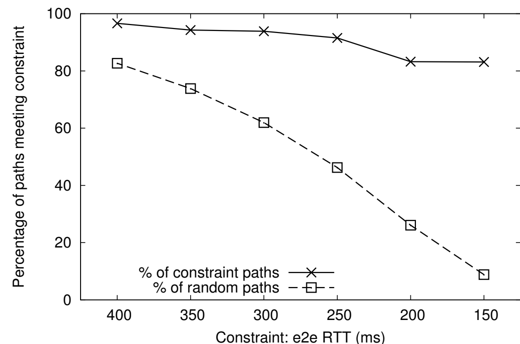

Figure 4. Achieved performance as measured by *e2e* RTT under simulation using the King dataset with the CONSTRAINT relay selection policy.

The performance of the anonymous paths shown in Figures 2 and 3 can be validated by comparing against results from previous studies [36] in which the relay selection algorithms were hardcoded. Using A^3, however, policies are concisely represented in a few lines of A^{3}LOG, and are provided to the Relay Selection Engine during runtime.

Our novel CONSTRAINT algorithm allows applications to specify hard constraints on their anonymous paths. The performance results for various e2e latency constraints is shown in Figure 4. The graph plots the percentage of anonymous paths whose e2e latency met the constraint for both the CONSTRAINT and RANDOM relay selection policies. The results from the uniform selection policy serve as an approximation for the percentage of possible paths that meet the constraint, and

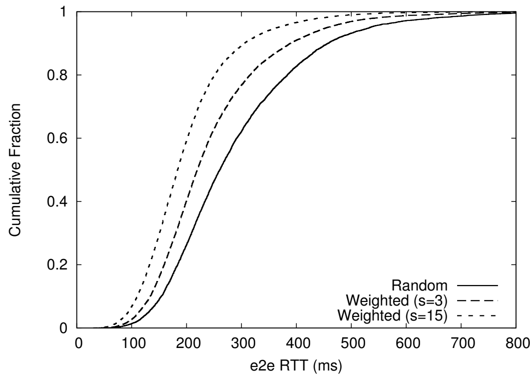

Figure 2. Achieved performance as measured by e2e RTT under simulation using the King dataset with the WEIGHTED relay selection policy.

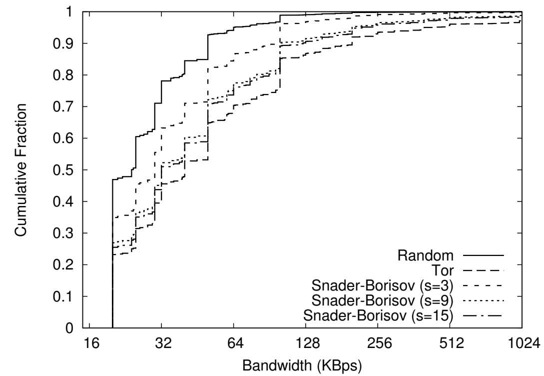

Figure 3. Achieved performance (log scale) as measured by e2e bandwidth under simulation using bandwidths from the Tor directory server for various node-based relay selection policies.

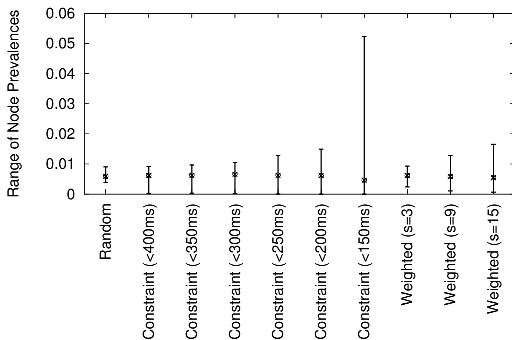

Figure 5. Node prevalences for latency datasets

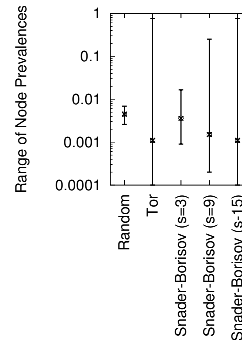

Figure 6. Node prevalences for bandwidth datasets *(log scale)***.**

therefore indicate the difficulty of finding conforming paths. Failure to meet the requirements specified by the CONSTRAINT strategy are due to embedding errors in the Vivaldi virtual coordinate system. That is, underestimations of network distances occasionally cause the Relay Selection Engine to incorrectly believe that a non-conforming path met the requirements of the policy.

When the constraint is lax and permits paths with e2e latencies of up to 350ms, 74% and 94% of the paths generated using uniform and CONSTRAINT, respectively, adhere to the requirement. Even for very stringent requirements – e2e latencies of 150ms or less – 83% of paths produced for the CONSTRAINT policy met the requirement. In contrast, less than 9% of random paths had latencies below the threshold.

In addition to enabling flexible routing, A^3 also serves as a tool for protocol designers to empirically measure some of the security characteristics of their algorithms. New protocols may be quickly encoded in A^3LOG and tested on A^3. The security of a given al-

gorithm may be partly assessed by examining the distribution of relays' *node prevalences* – the percentage of anonymous paths for which a given relay is a participant [36]. Comparing the node prevalences for various routing policies while keeping the network consistent provides a straightforward means of determining whether any particular relay is selected disproportionately during relay selection. Figures 5 and 6 plot the range of node prevalences for the previously described relay policies under simulation. Of particular interest is the *maximum* node prevalence – the percentage of paths that include the most popular chosen relay. As demonstrated in prior work [36], the node prevalences resulting from link-based path selection (left) tend to be significantly lower than that of node-based selection.

7.2.2 PlanetLab Deployment

To evaluate the system's performance on real-world networks, we installed A^3 on approximately 110 geographically distributed hosts on the PlanetLab testbed.

PlanetLab Performance. Figure 7 shows the e2e path performance results on PlanetLab for the RANDOM, WEIGHTED, and CONSTRAINT strategies. Due to instability in the PlanetLab network, paths were abandoned after a two second timeout, leading to a maximum RTT of 2s. WEIGHTED (with $s = 9$) reduced the median RTT of paths by 194ms (38%) as compared to random selection. 69% of paths met the fairly stringent 400ms requirement using the CONSTRAINT policy. By comparison, only 26% of random paths had e2e RTTs of less than 400ms.

Information Provider Polling Frequency In order to produce paths that adhere to application policies, the Routing Engine must rely on the data stored in the Local Directory Cache. If the data is stale, then routing decisions will be based on outdated information. However, frequent polling of the Information Providers consumes bandwidth both at relay nodes (whose resources may already be overburdened from forwarding traffic) and at the Providers. The rate at which information should be refreshed is highly dependent upon the particular metric. For example, bandwidth capacities may be fairly static, whereas bandwidth utilization varies significantly over time.

To understand this tradeoff for our Network Coordinate Information Provider, we examined the rate at which coordinates changed under high degrees of churn. Figure 8 (log scale on both axes) plots the rate of change (as measured by the distance between successive coordinate updates) on PlanetLab. Since relays operate independently and conduct coordinate updates at varying times, results are grouped at one minute intervals, with the 10th, 50th (median), and 90th percentiles plotted on the graph. Initially, 90% of all relays join the network at approximately the same time, resulting in substantial coordinate movement early in the experiment. However, the system quickly stabilizes– the median rate of change decreases to less than 10ms within 10 minutes. Hence, even in the near worst-case scenario in which all participants join the network at once, the coordinate system reaches equilibrium within approximately 10 minutes.

To model a more realistic scenario, the remaining 10% of PlanetLab nodes join the network after approximately 30 minutes (indicated by arrows on the graph). Immediately following the introduction of the new participants, the median difference between coordinate updates experiences a minor jump, but remains below 3ms.

Our results indicate that latency is fairly stable (at least on PlanetLab), requiring infrequent coordinate updates. Even when members of a large coalition of relays join the network simultaneously, the effect on coordinate stability is minor.

Processing Costs The Relay Selection Engine parses and interprets A^{3}LOG policies and uses the information stored in the Local Directory Cache to generate conforming paths. The engine is implemented in C++ as an extension to the P2 declarative networking system [19]. We observe in our simulations and PlanetLab experiments that the time required to parse A^{3}LOG scripts (which occurs only when such scripts are loaded) and produce paths is minimal. For example, when running on PlanetLab with a heavy network load, the Relay Selection Engine requires on average less than 200ms to produce a path using the CONSTRAINT policy with multiple constraints.

7.3 Path Instantiation

Above, our evaluation validated the flexibility of A^{3}LOG for supporting a wide range of relay selection policies with low performance overhead. Next, we benchmark the performance of declarative onion path instantiation. As described in Section 6, the A^{3}LOG implementation of this protocol requires the use of secure communication, as well as symmetric-key cryptographic primitives and onion assembly. To isolate the effects of CPU and communication overhead, we conducted our evaluation in a local cluster in addition to the PlanetLab testbed.

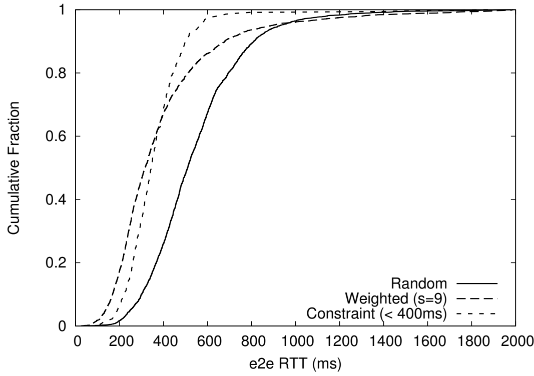

Figure 7. Achieved performance as measured by e2e RTT on PlanetLab.

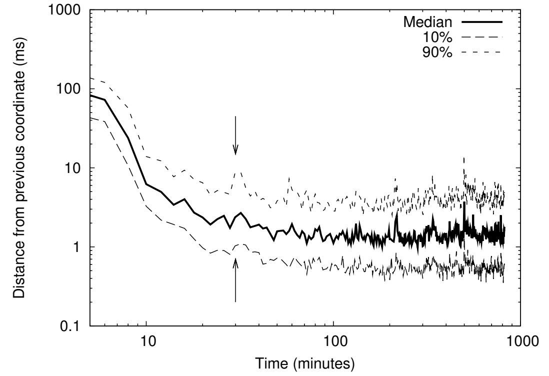

Figure 8. Median, 10th, and 90th percentile distances between coordinate updates on PlanetLab (log scale). Initially, 90% of all relays join the network at approximately the same time. Arrows indicate the point at which the remaining relays join.

Path Length	Local Cluster	PlanetLab
2	102	1059 (853, 1459)
3	146	1342 (1120, 2862)
4	192	2202 (1311, 3402)
5	244	2215 (1602, 2564)

Table 2. Median Onion Routing path instantiation time (in milliseconds) on our local cluster and on PlanetLab. The values in parentheses shows the 20th and 80th percentile times on PlanetLab.

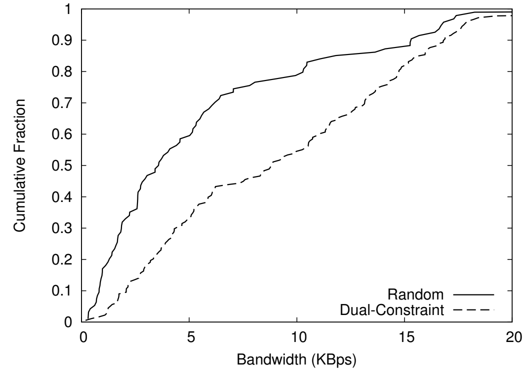

Figure 9. Distribution of bandwidth throughput on PlanetLab using the DJ-Anonymous application with a dual-constraint selection policy.

Our local cluster consists of quad-core machines with Intel Xeon 2.4GHz CPUs and 4GB RAM running Fedora 10 with kernel version 2.6, which are interconnected by Gigabit Ethernet. This setup allows us to isolate the computation overhead of onion routing in our benchmark.

Table 2 (second column) shows the path instantiation times (measured from the initiator creating the onion to the establishment of the bidirectional onion path) as the number of relays increases. For each relay size, we measured 10 path instantiations and computed the median. We make the following two observations: First, as expected, the path instantiation time increases linearly with the number of relays. Second, the instantiation time is within 244ms, even for up to 5 relay nodes, demonstrating the low overhead and efficiency of using A^3LOG.

Table 2 (third column) shows a similar experimental evaluation on the PlanetLab testbed, where we measure the median path instantiation times as the number of relays increases. We observe greater variability in path instantiation times (as shown by the values in parenthe-

ses denoting the 20th and 80th percentile) for the same number of relays. We further observe that on PlanetLab, the instantiation times are higher when the end-to-end latency dominates other factors due to the high load and network congestion observed on PlanetLab. Nevertheless, the majority of path instantiations complete within 2-3 seconds, even for many relays.

7.4 DJ-Anonymous A^3 Audio Streamer

To illustrate how applications can leverage the flexibility provided by A^3, we implemented a simple unicast audio streamer which we call *DJ-Anonymous*. DJ-Anonymous reads from an audio device at a fixed rate and transmits audio messages at regular intervals to the receiver. DJ-Anonymous supports the transmission of live audio and aims to minimize the latency and maximize the bandwidth of its chosen paths.

DJ-Anonymous uses a dual constraint policy in which the e2e latency is capped at 400ms and the tolerable maximum CPU utilization of a relay is set at 30% (as measured by CoMon [27]). Since rate limiting on PlanetLab prevents accurate bandwidth measurements, our policy does not include any bandwidth requirements. The entire policy used by DJ-Anonymous may be expressed in just six lines of A^{3}LOG.

In our experimental setup, each PlanetLab node runs an instance of DJ-Anonymous, sending streams of data to randomly selected PlanetLab destinations. Streams are instantiated on average every two minutes (stream start times are randomized between -20% and +20% to avoid nodes functioning in lockstep) and persist for one minute. Initiators attempt to send 500 byte payloads every 25ms, representing a maximum possible throughput of 20KBps. Since we were not able (or willing) to read from live audio sources on PlanetLab nodes, DJ-Anonymous instances on PlanetLab read "audio" from their `/dev/zero` devices.

Figure 9 shows the e2e bandwidth (as measured by the receiver) achieved using DJ-Anonymous' two-constraint relay selection policy. For comparison, the bandwidth that results from using random selection is also plotted. Using our six-rule dual-constraint policy, DJ-Anonymous achieves a median throughput of 9.0KBps (sufficient for G.711/PCM μ-law audio encoding), more than doubling the median bandwidth of 3.7KBps that results from random selection.

8 Conclusion

This paper presents Application-Aware Anonymity (A^3), an extensible anonymity framework that enables senders to compactly specify policies for relay selection and path instantiation that meet their performance and anonymity requirements. Unlike existing anonymity networks in which modifying the relay selection and path instantiation algorithms require changes to the source code, A^3 allows senders to customize the manner in which paths are chosen and constructed at runtime.

A^3 uses a declarative design in which senders specify their routing requirements using the A^{3}LOG policy language. We demonstrate that A^3 provides sufficient flexibility to encode the relay selection algorithms used by Tor [9], Snader and Borisov's refinement to Tor [40], and link-based approaches [36] in only a few lines of A^{3}LOG. Results from simulations over trace-driven datasets and our deployment on PlanetLab show that A^3 produces paths that conform to the specified policies with little computational overhead.

In addition to providing flexible relay selection, A^3 also enables initiators to customize both the manner in which anonymous paths are constructed as well as the mechanisms used to transport data over such paths. For example, we show how the Setup and Data Transmission phases of Onion Routing [30] can be compactly specified in A^{3}LOG.

A^3's flexibility has several advantages for anonymous routing. First, it allows senders to construct policies that meet their applications' specific requirements. For example, a real-time VoIP application may provide policies that enforce e2e latency constraints, whereas a file sharing client may utilize a policy that favors bandwidth over other performance indicators. Second, the ability to rapidly deploy both relay selection and path instantiation algorithms makes A^3 a useful tool for protocol designers and anonymity researchers. Finally, A^3's modular design and declarative techniques permit the system to be easily extended to support additional metrics. By constructing small adapters that interface with Information Providers, A^3 can be adapted to support policies that reference a diverse set of routing criteria.

Acknowledgments

The authors are grateful to the anonymous reviewers for their insightful feedback. This work is partially supported by NSF Grants CNS-0831376, CNS-0524047, and CNS-0627579; DARPA Grant ONR-N00014-09-1-0770; and OSD/AFOSR MURI *Collaborative Policies and Assured Information Sharing*. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

Appendix A. Link-based Tunable Performance/Anonymity Selection in A³LOG

The WEIGHTED link-based path selection algorithm operates in two phases:

In the first phase, the initiator rapidly generates (but does not instantiate) candidate paths consisting of three relays chosen uniformly at random without replacement. The initiator computes the e2e cost of each generated candidate path. In the second phase, the initiator sorts the candidate paths by their cost estimates. Using the family of functions introduced by Snader and Borisov [40] (see Eq. 1), the initiator instantiates the candidate path with index $\lfloor n \cdot f_s(x) \rfloor$, where x is chosen uniformly at random from $[0, 1)$, and n is the number of nodes. As with Snader and Borisov's algorithm, a larger value of s more heavily weighs path selection in favor of performance. The s parameter is denoted by the `S` attribute in the initial path request.

WEIGHTED is represented in A³LOG as follows:

```
w1 eCandidatePaths(Src, Dst, S,
                   RAND(3, 100)<IP,Coord>,
                   PathCosts) :-
     ePathRequest(Src, Dest, S), node(IP, Coord),
     Src != IP, Dst != IP, PathCosts = {}.

w2 eCandidatePaths(Src, Dst, S, PathList,
                   PathCosts) :-
     eCandidatePaths(Src, Dst, S, PathList,
                     PathCosts),
     f_size(PathList) > 0, P=f_popfront(PathList),
     PathCost=f_coorddist(Src.Coord,P[0].Coord) +
              f_coorddist(P[0].Coord,P[1].Coord) +
              f_coorddist(P[1].Coord,P[2].Coord) +
              f_coorddist(P[2].Coord,Dst.Coord),
     PathCosts.append([P, PathCost]).

w3 ePathResult(Src, Dest, Path) :-
     eCandidatePaths(Src, Dst, S, PathList,
                     PathCosts),
     f_size(PathList)=0,
     SortedPathCosts=f_sortByField(PathCosts,
                                   "PathCost",
                                   "desc"),
     sbRand=(1 - 2^(S*f_rand01())) / (1-2^S)),
     Path=f_selectIndex(SortedPathCost,sbRand).P.
```

Rule `w1` first generates 100 random permutations of three elements each from the node table. Then, rule `w2` repeatedly converts these list elements into pairs with the path's e2e cost, based on the embedded coordinates. Finally, rule `w3` sorts this list and selects an index using the Snader-Borisov random variable described in Eq. 1, with a tunable performance parameter `S`. Note that in this case, we sort in reverse order since lower latency is preferred to higher latency. The above rule assumes a left-to-right execution ordering of predicates. This assumption can be avoided with a more verbose version of the above program using some additional rules.

Appendix B. Tor and Crowds in A³LOG

Tor: Unlike Onion Routing, where the initiator recursively builds a single onion that is relayed along the entire path, Tor specifies an incremental telescoping path instantiation strategy. At a high level, a circuit initiator sends a *CREATE* message to the first *Tor router* in the desired circuit. The Tor router establishes local state and replies, resulting in a path of length one. Should the initiator choose to add another hop to the end of path, he relays an *EXTEND* message to the current endpoint. The current endpoint translates the EXTEND into a CREATE message and sends it to the desired new endpoint. The new endpoint of the circuit replies with a confirmation message, which is forwarded back to the initiator. The initiator may continue to extend the path if he desires[5].

Both CREATE and EXTEND messages can be encoded as A³LOG message tuples, and contain half of a Diffie-Hellman handshake, encrypted with the public key of the desired new endpoint. The new endpoint completes the handshake with the initiator, resulting in symmetric keys shared with the initiator, as in Onion Routing. The encryption/decryption modules can be implemented as a CView module with the corresponding cryptographic functions, similar to that described in Section 6.2. Tor's telescoping path establishment implies that n messages are exchanged in each direction to establish a circuit of length n. The kth message sent in the forward direction is essentially an onion with k layers. For sending messages between Tor routers, Tor specifies a link-layer protocol similar to that of Onion Routing.

Crowds: The process of path instantiation in Crowds commences when an initiator starts an anonymous relay on his machine called a *jondo* and contacts a server to obtain membership in a *crowd* – a collection of anonymous users. To build a path, the initiator forwards a request to a jondo chosen uniformly at random – possibly his own. Upon receiving a request to create a path, a jondo chooses to extend the path to another jondo (again chosen uniformly at random) with probability p_f, or ceases path creation with probability $1 - p_f$.

Typically, an initiator will use a single bidirectional path for all anonymous communication. However, in order to preserve anonymity properties, all initiators must create a new circuit – and cease using any previous ones – whenever a new jondo joins the crowd.

Below, we exhibit the forward half of the path instantiation scheme used in Crowds in the following rules:

[5] The current implementation of Tor uses three-relay anonymous paths by default.

```
c0 circuitStatus(CID, "BUILDING"),
   circuitForward(CID, ACI_out, Node_out),
   extend(@Node_out, ACI_out, Me) :-
       establish_path(), ACI_out=f_gen_aci(),
       CID = f_gen_cid(), random_jondo(&Node_out).
```

Rule `c0` begins the process of building a new path of jondos in response to an `establish_path` event. Such an event is triggered when a node retrieves a new list of jondos, for example. `c0` generates a CID and ACI for the new circuit, and selects a jondo uniformly at random (using the `random_jondo` CView) to receive the path extension request, `extend`. Upon receipt of an `extend` request, rule `c1` is triggered:

```
c1 circuitReverse(CID, ACI_in, Node_in),
   incoming(CID, X) :-
         extend(@Me, ACI_in, Node_in),
         X = f_rand01(), CID = f_gen_cid().
```

Rule `c1` generates a random number in the range $[0, 1]$, as well as a CID for the circuit. `c1` also derives a local `incoming` event, containing the local CID of the new circuit, and the previously generated random number. The `incoming` event triggers rule `c2`:

```
c2 circuitForward(CID, ACI_out, Node_out),
   extend(@Node_out, ACI_out, me) :-
     incoming(CID, X), p_forward(P), X <= P,
     ACI_out = f_gen_aci(),
     random_jondo(&Node_out).
```

Rule `c2` compares the random number (`X`) against the probability of extending the path forward, (`P`). If $X \leq P$, then rule `c2` generates an outgoing ACI, and selects a jondo uniformly at random to serve as the next relay in the circuit. Alternatively, if $X > P$, another set of rules relays a confirmation back to the initiator informing him that the newly instantiated path is ready for use.

References

[1] Tor Directory Protocol, Version 3. https://git.torproject.org/checkout/tor/master/doc/spec/dir-spec.txt.

[2] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. The Case for Resilient Overlay Networks. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 152, 2001.

[3] K. Bauer, D. McCoy, D. Grunwald, T. Kohno, and D. Sicker. Low-Resource Routing Attacks Against Tor. In *Proceedings of the 2007 ACM Workshop on Privacy in Electronic Society*, pages 11–20, 2007.

[4] O. Berthold, H. Federrath, and M. Köhntopp. Project “Anonymity and Unobservability in the Internet”. In *CFP '00: Proceedings of the Tenth Conference on Computers, Freedom and Privacy*, pages 57–65, 2000.

[5] M. Costa, M. Castro, R. Rowstron, and P. Key. PIC: Practical Internet Coordinates for Distance Estimation. In *International Conference on Distributed Computing Systems*, 2004.

[6] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A Decentralized Network Coordinate System. *SIGCOMM Comput. Commun. Rev.*, 34(4):15–26, 2004.

[7] F. Dabek, J. Li, E. Sit, F. Kaashoek, R. Morris, and C. Blake. Designing a DHT for Low Latency and High Throughput. In *NSDI*, 2004.

[8] R. Dingledine and N. Mathewson. Tor Path Specification, January 2008. http://www.torproject.org/svn/trunk/doc/spec/path-spec.txt.

[9] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-Generation Onion Router. In *Proc. of the 13th Usenix Security Symposium*, pages 303–320, 2004.

[10] H. Federrath. JAP: Anonymity & Privacy. http://anon.inf.tu-dresden.de/.

[11] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. IDMaps: A Global Internet Host Distance Estimation Service. *IEEE/ACM Trans. Netw.*, 9(5):525–540, 2001.

[12] M. Freedman, K. Lakshminarayanan, and D. Mazières. OASIS: Anycast for Any Service. In *Networked Systems Design and Implementation (NSDI)*, May 2006.

[13] M. J. Freedman and R. Morris. Tarzan: A Peer-to-Peer Anonymizing Network Layer. In *CCS*, Washington, D.C., November 2002.

[14] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating Latency between Arbitrary Internet End Hosts. In *ACM SIGCOMM Workshop on Internet Measurment (IMW)*, 2002.

[15] M. A. Kaafar, L. Mathy, C. Barakat, K. Salamatian, T. Turletti, and W. Dabbous. Securing Internet Coordinate Embedding Systems. In *ACM SIGCOMM*, August 2007.

[16] M. A. Kaafar, L. Mathy, T. Turletti, and W. Dabbous. Real Attacks on Virtual Networks: Vivaldi Out of Tune. In *SIGCOMM Workshop on Large-Scale Attack Defense (LSAD)*, pages 139–146, 2006.

[17] “King” Data Set. Available at http://pdos.csail.mit.edu/p2psim/kingdata/.

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

[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] H. Madhyastha, E. Katz-Bassett, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane Nano: Path Prediction for Peer-to-Peer Applications. In *Proc. of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2009.

[22] Y. Mao, B. T. Loo, Z. Ives, and J. M. Smith. MOSAIC: Unified Platform for Dynamic Overlay Selection and Composition. In *5th ACM International Conference on emerging Networking EXperiments and Technologies*, 2008.

[23] A. Mislove, G. Oberoi, A. Post, C. Reis, P. Druschel, and D. S. Wallach. AP3: Cooperative, Decentralized Anonymous Communication. In *11th Workshop on ACM SIGOPS European Workshop: Beyond the PC*, page 30, 2004.

[24] S. J. Murdoch and R. N. M. Watson. Metrics for Security and Performance in Low-Latency Anonymity Systems. In *8th Privacy Enhancing Technologies Symposium (PETS 2008)*, July 2008.

[25] T. S. E. Ng and H. Zhang. A Network Positioning System for the Internet. In *Proceedings of the 2004 USENIX Annual Technical Conference*, June 2004.

[26] L. Øverlier and P. Syverson. Locating Hidden Servers. In *IEEE Symposium on Security and Privacy*, 2006.

[27] K. Park and V. Pai. CoMon: A Monitoring Infrastructure for PlanetLab. http://comon.cs.princeton.edu.

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

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

[30] M. Reed, P. Syverson, and D. Goldschlag. Anonymous Connections and Onion Routing. *IEEE Journal on Selected Areas in Communications*, 16(4), May 1998.

[31] M. K. Reiter and A. D. Rubin. Crowds: Anonymity for Web Transactions. In *ACM Transactions on Information and System Security*, 1998.

[32] M. Rennhard and B. Plattner. Introducing MorphMix: Peer-to-Peer Based Anonymous Internet Usage with Collusion Detection. In *WPES '02: Proceedings of the 2002 ACM Workshop on Privacy in the Electronic Society*, pages 91–102, 2002.

[33] D. Saucez, B. Donnet, and O. Bonaventure. A Reputation-Based Approach for Securing Vivaldi Embedding System. In *Dependable and Adaptable Networks and Services*, 2007.

[34] Y. Shavitt and T. Tankel. Big-bang Simulation for Embedding Network Distances in Euclidean Space. In *IEEE Infocom*, April 2003.

[35] M. Sherr. *Coordinate-Based Routing for High Performance Anonymity*. PhD thesis, CIS Department, University of Pennsylvania, 2009.

[36] M. Sherr, M. Blaze, and B. T. Loo. Scalable Link-Based Relay Selection for Anonymous Routing. In *9th Privacy Enhancing Technologies Symposium (PETS 2009)*, August 2009.

[37] M. Sherr, M. Blaze, and B. T. Loo. Veracity: Practical Secure Network Coordinates via Vote-based Agreements. In *USENIX Annual Technical Conference (USENIX '09)*, June 2009.

[38] M. Sherr, B. T. Loo, and M. Blaze. Towards Application-Aware Anonymous Routing. In *Second USENIX Workshop on Hot Topics in Security (HotSec)*, August 2007.

[39] C. Shields and B. N. Levine. A Protocol for Anonymous Communication over the Internet. In *CCS '00: Proceedings of the 7th ACM Conference on Computer and Communications Security*, pages 33–42, New York, NY, USA, 2000. ACM Press.

[40] R. Snader and N. Borisov. A Tune-up for Tor: Improving Security and Performance in the Tor Network. In *15th Annual Network and Distributed System Security Symposium (NDSS)*, February 2008.

[41] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *SIGCOMM*, 2001.

[42] B. Wong, A. Slivkins, and E. G. Sirer. Meridian: a Lightweight Network Location Service without Virtual Coordinates. In *SIGCOMM*, 2005.

[43] D. J. Zage and C. Nita-Rotaru. On the Accuracy of Decentralized Virtual Coordinate Systems in Adversarial Networks. In *CCS*, 2007.

[44] L. Zhuang, F. Zhou, B. Y. Zhao, and A. Rowstron. Cashmere: Resilient Anonymous Routing. *Proc. of NSDI*, 2005.