NetQuery: A General-Purpose Channel for Reasoning about Network Properties

Alan Shieh Oliver Kennedy Emin Gün Sirer Fred B. Schneider

Dept. of Computer Science, Cornell University, Ithaca, NY 14853
{ashieh, okennedy, egs, fbs}@cs.cornell.edu

ABSTRACT

Although the configuration of modern networks has a significant impact on the performance, robustness, and security of applications, networks lack support for reporting these differences. This paper presents the design and implementation of NetQuery, a novel, general-purpose channel for disseminating the properties of networks and their participants. NetQuery implements a distributed, decentralized, tuple-based attribute store that records information about network entities. Operators can add new tuples into this store and can also annotate existing tuples with new, custom attributes, thus allowing the system to support network entities and properties not anticipated at the time of deployment. NetQuery clients can query this attribute store for the current network state and install event triggers to detect future state transitions, thus establishing long-running guarantees over the behavior of the network. We have implemented NetQuery and deployed networks with NetQuery-enabled devices that leverage commodity trusted hardware to provide strong assurance over the accuracy of reported properties. We describe the NetQuery system, outline the types of new applications enabled by NetQuery, and report on the performance of the system from deployments of real devices and from simulations of ISP networks.

1. INTRODUCTION

Depending on their configuration, administration and provisioning, networks can provide drastically different features. For instance, some networks provide little failure resilience while others provision failover capacity and deploy middleboxes to protect against denial of service attacks [4]; some networks provide network neutral access while others selectively throttle applications [12]; and some networks provide confidentiality guarantees while others monitor a user's request stream to build an advertising profile [17]. Yet the standard IP interface masks these differences, as every network appears to provide the same basic "dial-tone" service. Consequently, clients resort to ad hoc techniques to detect these differences or target the lowest common denominator service.

Similarly, configuration differences between end hosts are of interest to network operators, yet network operators lack a channel for securely querying the properties of their clients. For instance, hosts configured without firewalls and virus checkers can launch internal attacks on a protected network, while hosts with untrusted network stacks can monitor network traffic or consume excess network capacity. Network administrators often resort to expensive manual scans to check their clients' configurations [15]. Similarly, lack of a standard channel to the user leads ISPs to try to communicate with the user through disruptive mechanisms such as captive portals and interstitial web pages. These mechanisms are brittle, complex and can break client applications.

This paper describes NetQuery, a general-purpose channel for distributing and reasoning about the properties of network entities. By network entities, we mean both the physical devices, such as routers, switches, and end hosts, and the virtual entities, such as flows and applications, that comprise a network. Each network entity is associated with a set of properties, represented in NetQuery as unconstrained attribute/value pairs. These properties may be intrinsic to a device, such as a router's routing tables, or they may be arbitrary labels assigned by third parties, such as a certificate from an audit service that a router is properly configured. NetQuery implements a distributed, decentralized tuplestore that stores the attribute/value pairs of network entities. The tuplestore provides a query interface through which applications can retrieve the properties of network entities and a trigger interface through which applications can detect changes to these properties. These interfaces support the establishment of long-running guarantees about the network: the query interface enables applications to deduce that a guarantee holds for an initial network state, while the trigger interface enables applications to detect and modify, where applicable, network changes that can invalidate the guarantee.

While NetQuery provides a globally unified interface, the implementation and deployment of the tuplestore is decentralized. Every network operator deploys a NetQuery server dedicated to storing the attribute/value pairs for its portion of the network and specifies an access policy to these attribute/value pairs. NetQuery enables anyone to tag any entity with an arbitrary property without the need for a central administrator to mediate conflicts.

Although the tuplespace may contain conflicting information from different sources, NetQuery enables applications to identify information from sources that they trust. The tuplespace records for each attribute the principal, such as a device, system administrator, or audit service, that generated it. A policy language enables applications to establish trust in the attribute/value pairs stored in NetQuery, and to control access to proprietary information and sensitive operations.

Reasoning about properties of remote network entities only makes sense when there is an established basis to trust their claims. NetQuery is based on a "clean-slate" design where all networked components are assumed to be equipped with secure hardware coprocessors; this trusted hardware serves as a root of trust for claims about the network. Coprocessors such as the Trusted Platform Module (TPM) are cheap and becoming ubiquitous [6]. Such secure hardware makes it possible to issue unforgeable certificates describing the software environment and dynamic state of a device through a mechanism known as *attestation* [8]. While our work focuses on translating such local guarantees into network-level assurance, we also discuss how current systems equipped with management interfaces such as SNMP but no secure coprocessor might evolve towards supporting NetQuery services. The base system design supports incremental deployment, though such a deployment necessitates a larger trusted computing base (TCB).

Three sample scenarios illustrate the types of applications that NetQuery enables:

- *Checking end hosts.* Misconfigured end hosts can compromise the integrity of a network. A NetQuery-enabled network can restrict access based on the configuration of end hosts. For instance, before allowing a newly connected end host to send packets, a switch can verify that the end host is running the latest software versions and a virus checker.

- *Checking paths.* Middleboxes that can perform sophisticated deep packet inspection for large volumes of network traffic enable ISPs to monitor and modify streams that traverse their network. A privacy conscious user who wants to know how data from her web sessions will be monitored can use NetQuery to discover whether there are entities that can potentially access and record her sessions and, if applicable, to obtain guarantees from the network on how packets are handled.

- *Differentiating providers.* Customers currently differentiate between ISPs based solely on reputation. NetQuery enables ISPs to advertise the performance and robustness features that they provide. For instance, a wireless service provider can use NetQuery to advertise its backhaul capacity and traffic management techniques; clients can use this to select the best available network.

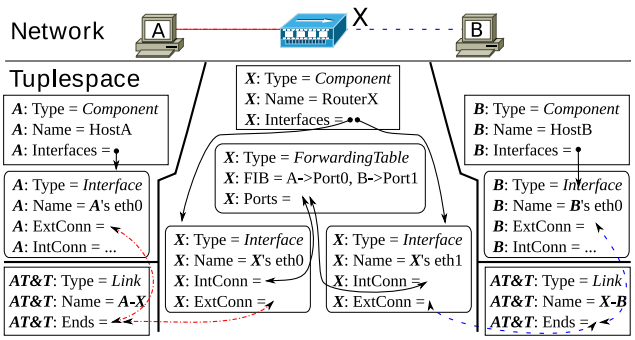

Figure 1: **A simple network and its tuplespace representation.** Every network entity has a corresponding set of tuples that describes its current configuration and state. Attribute/value pairs are prepended with the speaker name.

This paper outlines the case for NetQuery and makes three contributions. First, it proposes a set of abstractions for disseminating network properties and outlines a realizable distributed architecture for storing meta-information about network entities. Second, it proposes a representation for network entities and their properties and shows, by way of example applications, how analysis over this representation enables novel applications. Finally, it shows that NetQuery scales to the query load from millions of clients and to the volume of properties needed to represent the devices of a large network.

This paper is structured as follows. Section 2 presents the NetQuery abstractions and interfaces. Section 3 describes how applications establish the verity of properties, and how information providers control access to properties. Section 4 provides example applications. Section 5 describes our implementation, and Section 6 reports the performance of the system. Section 7 presents related work, and Section 8 concludes.

2. THE NETQUERY TUPLESPACE

The central abstraction for disseminating network information in NetQuery is that of a *tuplespace* (Figure 1). The tuplespace presents to applications a tuple-based representation of the network that reflects its topology and configuration and describes every network entity with a corresponding set of tuples that describes its external connections, internal configuration, and runtime state. In addition to the basic access interfaces for storing and retrieving data, the tuplespace provides attribution and extensibility for tuples and can delegate its operation to different administrative domains, that together make it well-suited for representing heterogeneous networks such as the Internet.

Every NetQuery entity accesses the tuplespace as a *principal*. A principal is a user that is granted access privileges to the tuplespace. Every principal is associated with a unique public/private key pair. This key acts as an identifier for the

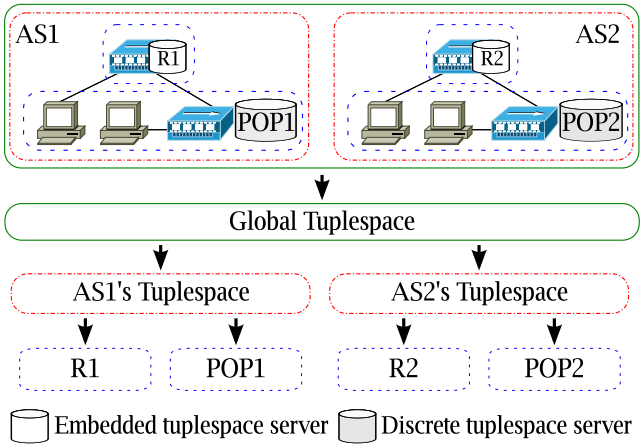

Figure 2: **NetQuery tuplespace delegation.** Even though NetQuery presents a unified tuplespace, the implementation is decentralized across multiple administrative domains and distributed across multiple servers, including those embedded in network components.

principal, without the need to resort to PKI certificates to provide identities and is used for authentication.

NetQuery provides a single global tuplespace for the Internet that is distributed across multiple tuplespace servers. The responsibility for managing *subspaces*, disjoint subsets of the tuplespace, is delegated to multiple administrative domains (Figure 2). Every AS or third party information service manages a subspace covering its own network and annotations. An AS or information service is responsible for deploying tuplespace servers for storing their local tuples and is empowered to define an access policy for the tuples within their subspace.

2.1 Data model and representation

Every tuple is named by an opaque, globally unique *tuple ID* (TID) and stores data as typed *attribute/value pairs*. NetQuery provides scalar primitives, references, and aggregate attribute types. Integers and strings comprise the scalar primitives. References contain pointers to NetQuery tuples, encoded as tuple IDs, and are used to chain NetQuery tuples into complex data structures such as trees or graphs. Aggregate types, such as vectors, sets, and tries, are used to represent information of varying cardinality, such as the forwarding entries used during routing. Applications and devices create, read, modify, and delete tuples and attribute/value pairs using primitive operations exported by the tuplespace.

Tuples conform to *schemas*, which define the set of attribute/value pairs that all tuples for a given kind of network entity must provide. The schema thus establishes a common baseline representation that enables programmers to write analysis code over the various network entities. A schema differs from a class in object systems in that it describes just this common set of attribute/value pairs and not the code or operations. The schema name of an tuple is stored within

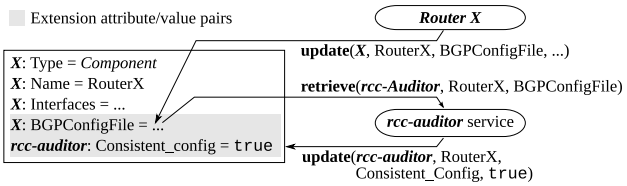

Figure 3: **NetQuery extensibility and annotations.** A BGP NetQuery router has extended the base router representation with information about its BGP configuration. A third-party audit service has annotated router ***X*** after checking its configuration.

the *tuple_type* attribute of the tuple. NetQuery clients use this field to determine how to interpret the attribute/value pairs contained in a tuple.

NetQuery provides built-in schemas for representing *processing entities*, which describe the devices or software endpoints that generate, receive, or forward packets; and for representing *protocol entities*, which describe the higher-level conversations between processing entities. Every packet is associated with one or more protocol entities. Processing entities include hosts, routers, switches, network cards, and applications. Protocol entities include TCP connections, TCP/UDP endpoints, and IPsec security associations.

These schemas together enable a comprehensive description of all network entities that covers their hardware and software configuration. The tuples for processing entities describe their operation in terms of their functional blocks, such as switching fabrics, network interfaces, network links, and packet filters. Tuples with this generic representation are not restricted to describing only existing network entities and can be used to extend NetQuery with support for new kinds of network components. The schemas for protocol constructs describe the protocol state and end points.

Any NetQuery speaker, whether the creator of a tuple or a third party, can extend a tuple with additional attribute/value pairs, called *annotations* (Figure 3). For instance, a router that implements BGP can annotate the base router tuple with BGP configuration information. A third-party audit service can run a configuration analysis tool such rcc [7] to annotate a router with the property

$$consistent\_config = \texttt{true}$$

asserting that it has been configured to propagate consistent routes.

2.2 Initialization and state maintenance

Every network device present in the network corresponds to a tuple. The tuples and their attribute/value pairs can be initialized and maintained by different parties, including the device itself, its administrator, or a third party.

In clean-slate deployments, every NetQuery device initializes its own tuplespace representation and is configured with a local NetQuery server. On start up, a device issues

create() operations to this server to instantiate the tuples that represent it, and issues **update**() operations to set the attribute/value pairs based on its local configuration and initial state. For instance, a freshly activated router creates tuples for all of its interfaces and exports its initial forwarding table state. A newly connected endpoint creates tuples for each of its interfaces and defines attribute/value pairs describing its system statistics such as OS version, network services provided, or any security guarantees obtainable from trusted hardware. Incremental deployments, wherein not all devices embed NetQuery speakers, use management services, such as SNMP-to-NetQuery proxies, to export the same per-device information.

2.3 Queries

NetQuery applications use the contents of the tuplespace to derive conclusions about network devices, processes, and the network itself. In principle, NetQuery applications use the entire tuplespace state as the basis for these conclusions. In practice, applications perform tuplespace queries to find and retrieve the relevant subset of this state and install triggers to detect any network changes that might impact the conclusion. The process of inferring some high-level *characteristics* about the network, such as the route between two hosts, from low-level properties, such as routing tables, is called *analysis*. Together, the combination of queries and triggers can check a range of properties that varies in breadth of network coverage and duration of validity.

Clients discover tuples of interest by issuing **query**() to the tuplespace. **query**() accepts a tuple pattern as a parameter and returns the TIDs of tuples that match this pattern. Tuple patterns consist of boolean combinations of predicates over the contents of a tuple. These predicates specify identity and wildcard matches for attribute values, tuple IDs, and principal names. For instance, the query

$$tuple\_type = \mathsf{Router}\ \texttt{AND}\ zip\_code = \texttt{10001}\ \texttt{AND}\ tid = \texttt{AT\&T} :: *$$

returns the tuple IDs of all routers at a particular geographic region, restricted to those tuples describing AT&T's network. NetQuery executes queries on tuplespace servers, enabling clients to efficiently search for tuples without retrieving large portions of the tuplespace. The pattern language specifically excludes operations such as allocation, loop constructs, and recursion, that can consume unbounded amounts of memory and computation. Hence, queries are safe to execute on tuplespace servers.

Clients issue **retrieve**() to access tuples whose tuple IDs they already know. This operation takes a tuple ID and a set of attribute names, specified as an attribute name pattern, and returns the corresponding attribute values from the specified tuple if they exist. The operation returns an error if the attribute/value pair does not exist. A client might acquire tuple IDs for use in **retrieve**() from the reference-typed attribute/value pairs of another tuple, or receive them out of band.

Applications vary in how they derive properties and in the consistency guarantees that they need. Some properties derive from the value of a single attribute. For instance, a physical link is either wired or wireless and a single query suffices to return the relevant attribute. Other properties derive from multiple attribute/value pairs. For instance, NetQuery can determine the route to a destination by iterating through routing tables: each iteration retrieves the next hop router for the next iteration.

When multiple attribute values are requested by an application, NetQuery returns a recent value for each. This consistency between multiple attribute values is analogous to that provided by sampling the network state with probes, such as by sending packets to perform traceroute or bandwidth measurement. Applications typically send multiple packets for a characteristic: for instance, in traceroute, a packet is sent for every hop on a path. Similarly, NetQuery analyses based on multiple attribute/value pairs generally retrieve these with a sequence of queries. Probes reveal information about the network as packets traverses it and thus only measure a particular slice of the network at a particular point of time. Should the network change during probing or attribute retrieval, both the probe- and attribute-based analyses will potentially use inputs based on inconsistent views of the network, resulting in an analysis result corresponding to a state the network never actually entered. NetQuery's consistency is appropriate for existing applications for which the level of consistency provided by sending probe packets suffices. NetQuery applications benefit from a programming model that enables them to directly extract, rather than infer, low-level network properties through attribute/value pairs, reserving programmer effort for deriving higher-level characteristics.

NetQuery also enables new applications that require stronger consistency guarantees. NetQuery applications that depend on long-running characteristics can use triggers to react to network changes that might invalidate a characteristic. For instance, suppose an access control agent allows LAN access only to end hosts that run approved software. To discover these hosts, the access control agent issues queries that traverse the LAN's topology graph. If the graph changes during this traversal, the access control agent can miss new end hosts. These missed hosts cannot connect to the network, since the access control agent will never examine them to authorize access. To discover all hosts, the access control agent can install triggers to detect topology changes and restart the analysis as needed.

The remainder of this section describes the tuplespace abstractions that enable applications to detect, react to, and control network changes; and to perform analyses that span multiple attribute/value pairs.

2.3.1 Detecting and controlling network changes

Changes to the network state are propagated to the tuplespace through corresponding *events*. These changes can

invalidate a property that was previously detected by an application. NetQuery applications use triggers to monitor, or to maintain as valid, a network characteristic over time. A trigger registers an application's interest in detecting and sometimes modifying events. In some cases, triggers can warn applications when a property on which they relied is about to change. In others, triggers can maintain a property on behalf of an application by responding to network events with remedial actions. Thus, triggers enables guarantees to clients of the form, "Either characteristic X holds, else notify client C."

A trigger consists of a predicate, a trigger notification type, and the IP and port number of a recipient. The predicate is written in the same pattern language as $\mathbf{query}()$ invocations and specifies a set of attribute/value pairs that are monitored for intended updates. Changes that match the predicate cause the trigger to *fire*: when a trigger fires, NetQuery sends a notification packet describing the change to the indicated IP/port; this packet describes which tuple and attribute/value pairs have changed, and has the semantics of a remote upcall to the trigger recipient. Since time is defined relative to tuplespace changes, NetQuery servers do not require tightly synchronized clocks.

NetQuery does not run any application-specific code during trigger processing: it pushes the bulk of custom update processing to clients, leaving the tuplespace responsible only for update detection. This reduces the processing overhead and complexity of NetQuery servers. NetQuery clients can use triggers to change how NetQuery updates the tuplespace in response to a modification request; thus, NetQuery interprets these requests as an intent to update.

The controllability of a change determines what actions can be taken when a trigger fires. Some changes cannot be controlled by software or hardware. For example, if a user physically unplugs a network cable, the resultant changes to the tuplespace cannot be averted. These changes give rise to *advisory* events. Often, these real-world changes result in collateral events that can be controlled. With NetQuery, these can be deferred, or aborted before they take effect. NetQuery exposes to applications this control over events and network devices. These are *delayable* or *vetoable*. For example, while NetQuery has no control over when a user plugs a computer into a switch, the switch does not need to immediately provide full network connectivity to the computer. Enabling full network connectivity is both delayable and vetoable, provided the switch can be instructed to restrict connectivity. Networks can enforce policy by deferring such events so that they can take action before the event occurs. A topology management application can delay an end host's request to attach to a switch so that it can first reconfigure the switch to place the new end host within the correct VLAN. Similarly, a firewall can expose insertion of new TCP firewall connection table entries for new flows as a vetoable event. An access control agent can intercept such updates to verify that a new flow is allowed by local policy.

Applications can detect events to react to network changes. Suppose an application uses a particular route because that route exhibits a desired characteristic. A routing table change might alter the route, changing this characteristic. For instance, applications that use NetQuery to discover high-capacity routes for improving throughput, can use advisory events to adjust its optimization strategy, while applications that use NetQuery to discover routes with stronger confidentiality guarantees, such as those that lack unencrypted wireless, can use delayable events give the application the opportunity to check that the new route is acceptable before switching to it.

Some events are potentially controllable but difficult to make so under current device architectures. For example, forwarding table updates are fully under software or hardware control, and it is straightforward to make them delayable, because delay can be injected with largely local code changes. However, making such events vetoable requires error handling code to react to vetoed updates.

The notification type determines what a recipient can do in response to the intended update. There are three types, *ask-veto*, *ask-delay*, and FYI; ask-veto allows the recipient to cancel the update, ask-delay allows the recipient to defer the update from taking effect, up to a maximum delay, while it takes corrective action, and FYI (for your information) serves as a one-way notification for a change in the network. In FYI notifications, information is only pushed to the recipient, while for ask-veto and ask-delay, the server waits for the recipient to respond with whether to veto or delay the update.

NetQuery servers mediate between tuplespace triggers and the network by forwarding a veto or delay to the affected network devices. Thus, vetoes or delays result in canceled or delayed changes to device state, and hence behavior. The allowable notification types vary, depending on the type of event that generated the intended tuplespace update and on the policy of the affected network entity. Vetoable events support all trigger types, delayable events support ask-delay and FYI, and advisory events support only FYI.

2.4 Tuplespace access control

Exporting information about network entities can cause privacy violations, while allowing triggers to exert control over events can allow malicious users to cause a network to malfunction. For instance, network operators might want to protect proprietary information by restricting who can access information about their internal networks, and prevent untrusted users from disrupting network connectivity by restricting ask-veto or ask-delay triggers on routing updates. Similarly, describing an end host's software configuration can reveal private activities about a user. To address these privacy and safety concerns, NetQuery associates every attribute/value pair with a policy provided by its creator that specifies which NetQuery clients are allowed to read, query, and install triggers on that attribute/ value pair (Section 3).

2.5 NetQuery servers and access protocol

NetQuery servers export a tuplespace interface to NetQuery clients and speakers. Every NetQuery principal is associated with an administrative domain, whose tuplespace servers process queries and store tuples and attribute/value-pairs on behalf of its local principals. Requests that reference a tuple or attribute/value-pair are transparently routed to the tuplespace server that stores it; this is achieved by storing within every tuple ID and attribute name a pointer to this "home" server, as identified by the administrative domain's name and the tuplespace server's IP and port. The remainder of the TID is typically a unique, opaque byte string.

Some queries, such as those that use wildcard patterns, require accessing multiple attribute/value pairs from a tuple. Since the storage location of each attribute/value pair may reside in tuplespace servers operated by different administrative domains, multiple servers may need to be contacted to process such queries. Such queries are sent to the server that stores the tuple, which is responsible for generating a query execution plan. To determine which other servers to contact, the server stores a forwarding pointer to every remote tuplespace server that stores one or more attribute/value pairs for the tuple.

A tuplespace server may be embedded within a network component, such as a router, that exports network information, or it can operate as a discrete server in its own right. Tuplespace servers employ lease-based storage management for tuples and periodically purge tuples that have not recently been touched.

3. POLICIES FOR TRUST AND ACCESS CONTROL

Managing the diversity of trust relationships is a central challenge in disseminating network properties in a heterogeneous network. Applications and information providers can specify policies to control how information is (1) imported and (2) exported across trust boundaries, and (3) to control access to potentially dangerous tuplespace operations.

These three different contexts all share a common policy language establishing trust in network entities. In NetQuery, every network entity is associated with a principal. Every *utterance*, or attribute/value pair or tuplespace operation issued by an entity, is attributed to that entity's principal. The principal to which an attribute/value pair is attributed is stored in the tuplespace along with the value; this attribution information is returned along with the value on retrieval. In NetQuery, principals are bound to a private keys. In principle, every utterance is signed by the key corresponding to the speaker; in practice, these keys are used to establish secure channels that serve as efficient surrogates for signatures.

Every NetQuery policy evaluation reduces to determining whether an entity trusts some utterance from a principal. For instance, suppose an end host wants to characterize a route. This analysis traces through the tuplespace for routing information. The accuracy of this conclusion is contingent on the accurate of the inputs: if some analysis were to rely on a routing table tuple that was exported from a malicious router, and that router did not faithfully copy its real routing table to the tuplespace, then the inferred route could differ from the real route. To protect against this, the end host would specify import policy that accepts only information from trusted routers, such as those equipped with TPMs.

A NetQuery policy is a set of statements that describes a set of trusted utterances. Policy statements are based on delegation. In delegation, a principal `A` empowers another principal `B` to make statements on its behalf. Delegations are expressed in policy statements of the form

$$\texttt{A says B speaksfor A on } \textit{subjects}$$

The *subjects* clause is a pattern that describes the scope of the delegation. These patterns are expressed in terms of tuple IDs, attribute names, and operations, and are matched against utterances. The NetQuery policy language supports restrictions based on a tuple's schema, owner, and underlying device. Delegation restrictions add a layer of indirection that improves convenience and efficiency while preserving security. Delegation is transitive: a principal that is empowered to make statements about some subject is also empowered to delegate this right to others. NetQuery's policy semantics allows only a narrowing of subjects on each delegation, thus preventing principals from using delegation to escalate their privilege.

Consider the import policy for a route characterization application `Client`. With an empty import policy, `Client` only trusts statements from itself. To trust statements from a router $\texttt{R}_0$, `Client` specifies the import policy

$$\texttt{Client says R}_0 \texttt{ speaksfor Client on } \textit{Router}.*[\textit{router\_id} = R_0]$$

Rather than directly vet every router in the world, which is cumbersome, inefficient, and unrealistic, the client instead delegates the responsibility to the manufacturer.

$$(1)\ \texttt{Client says Cisco speaksfor Client on } \textit{Router}.*$$

The first statement asserts that the client trusts Cisco to make statements about routers. To empower a router to make statements about itself, Cisco ships it with a unique public/private key pair `X` and publishes the delegation certificate, which restricts the router to only making statements about itself.

$$(2)\ \texttt{Cisco says X speaksfor Cisco on } \textit{Router}.*[\textit{router\_id} = X]$$

When the client retrieves a routing table $\textit{Router.routing\_table}$ attributed to `X`, the combination of statements (1) and (2) prove that this routing table is trusted.

3.1 Evaluating policies

Policies are not necessarily static: they can depend on changes to the network. For instance, the set of routers that is trusted can grow as new routers are installed. Thus, it is not possible to include all statements such as (2) in a policy, since doing so requires enumerating every possible router.

NetQuery provides mechanisms for publishing and discovering such statements. Statements are published to the tuplespace by the issuing principal like any other property. When a client checks a policy, it locates the relevant statements to use as credentials. There are several possible mechanisms for doing so. For instance, a proof checker can automatically search the tuplespace for the necessary credentials. While this is convenient for the programmer, this search can be costly to execute. Often, policies follow an established pattern, wherein all necessary credentials are in well-known attribute/value pairs. For the router policy, every router stores the delegation statement issued by Cisco in the attribute/value pair *Router.credentials*. Thus, a general search is not needed.

NetQuery supports both of these mechanisms. NetQuery provides a schema for exporting policy statements as tuples. Proof checkers can use tuple queries to search for relevant statements, just as they do for other tuples. The metadata stored with a tuple or attached to a tuplespace operation embeds credentials supplied by the originating principal; these credentials are passed to the recipient.

3.2 Increasing policy expressiveness with trusted computing

Entities built using trusted computing primitives can export detailed and trustworthy descriptions of themselves to the tuplespace. TPMs form the basis for establishing trust in these self-reported properties. TPMs generate certificates, called attestations, that describe a chain of trust, rooted at a TPM, where each link represents a layer of hardware or software that is responsible for identifying and vetting the next layer. For instance, the TPM vets the trusted OS, which in turn asserts that a process with a given hash and configuration is running. TPMs provide a *Quote* operation that cryptographically binds a message to an attestation. Trusted operating systems can use quote to export detailed information about its processes that can be trusted by remote hosts. Quoted information is translated to NetQuery attribute/value pairs by deriving the principal from the attestation and the attribute/value pair from the message.

This information expands the expressiveness of policies. For instance, rather than identifying trusted entities by principal name, policies can instead identify trusted entities by their properties.

Consider the tuple representation of a TPM-equipped host running a trusted Linux and a software router (Figure 4):

(1) The trustworthiness of the host hardware platform to make statements about the OS is established by the boot-time attestation, stored in the *Host.credentials* field. The trustworthiness of the OS to make statements about its processes

(1) $\texttt{TPM}_\texttt{i}.\texttt{Linux}_\texttt{j} = \langle$
[$\texttt{TPM}_\texttt{i}$, *tuple_type* = `Host`],
[$\texttt{TPM}_\texttt{i}$, *credentials* = `TPM HW platform attestation`],
[$\texttt{TPM}_\texttt{i}$, *hash* = `TrustedLinuxv2.6`],
[$\texttt{TPM}_\texttt{i}.\texttt{Linux}_\texttt{j}$, *process* = $\texttt{TPM}_\texttt{i}.\texttt{Linux}_\texttt{j}.\texttt{SoftRouter}_\texttt{k}$]$\rangle$
(2) $\texttt{TPM}_\texttt{i}.\texttt{Linux}_\texttt{j}.\texttt{SoftRouter}_\texttt{k} = \langle$
[$\texttt{TPM}_\texttt{i}.\texttt{Linux}_\texttt{j}$, *tuple_type* = `Process`],
[$\texttt{TPM}_\texttt{i}.\texttt{Linux}_\texttt{j}$, *hash* = `Routerv1.0`],
[$\texttt{TPM}_\texttt{i}.\texttt{Linux}_\texttt{j}$, *allow_only* = `CfgRTNetlink`],
// allow_only of all other processes exclude CfgRTNetlink
$\rangle$
(3) $\texttt{R1} = \langle$
[$\texttt{TPM}_\texttt{i}.\texttt{Linux}_\texttt{j}.\texttt{SoftRouter}_\texttt{k}$, *tuple_type* = `Router`],
[$\texttt{TPM}_\texttt{i}.\texttt{Linux}_\texttt{j}.\texttt{SoftRouter}_\texttt{k}$, *routing_table* =
$\{10.0.0.0/8 \Rightarrow \texttt{eth0}, \ldots\}$]$\rangle$

Figure 4: **Tuple representation for TPM-equipped software router.** Per-attribute/value pair credentials are omitted.

is established by *Host.hash*. (2) Even though the client trusts the `Routerv1.0` software to properly report routing information, checking the hash is not sufficient. Under Linux, any sufficiently privileged processes can modify the kernel routing table by invoking the RTNetlink kernel interface, causing the device state to diverge from the tuple representation. Thus, the import policy for router information (3) verifies that (a) the router process has the right hash (*Process.hash*) and (b) only the router process, and no other process, can invoke RTNetlink (*Process.allow_only* of all processes).

3.3 Summary

Figure 5 summarizes how policies are integrated into the NetQuery interface and data model.

3.3.1 Import policies

Client applications check import policies for all inputs received from the tuplespace (Figure 5-B.1). These checks cover both values retrieved from the tuplespace and trigger notification upcalls, which deliver information about value changes in attribute/value pairs. Import policies are typically specified by the application developer, but can be modified by users according to local requirements. For instance, a system administrator might relax policies to trust all statements made by routers in the intranet.

3.3.2 Export policies

The tuplespace server checks all operations that can return tuples and attribute/value pairs against export policies (Figure 5-B.2). In addition to query and retrieve, which immediately return information, trigger notification upcalls are also checked, since they return information in response to network changes. Export policies can be specified for every

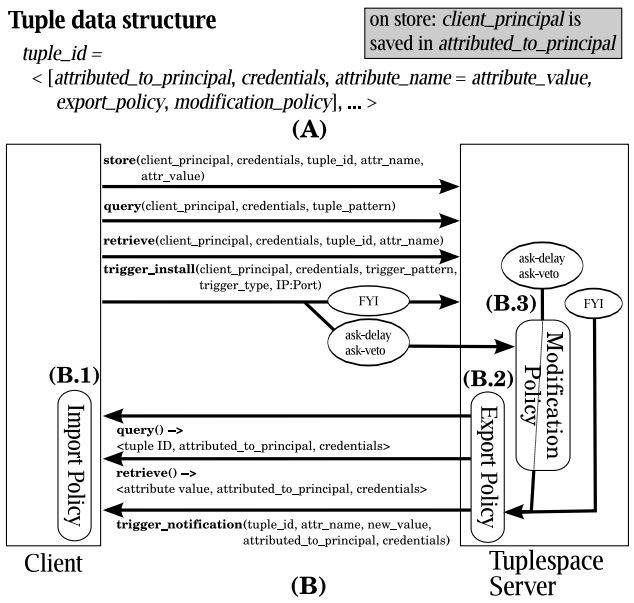

Figure 5: **Complete NetQuery architecture.** (A) Extensions to data model to support attribution and credentials. (B) Location of policy enforcement: (B.1) Import policies. (B.2) Export policies. (B.3) Modification policies.

attribute/value pair, and are typically specified by the user. For instance, an ISP that wants to protect proprietary information from escaping its network would allow NetQuery access only from entities residing on its own network.

3.3.3 Modification policies

The tuplespace server checks against modification policies any operations, such as ask-delay and ask-veto trigger installation and notification upcalls, that can modify the behavior of the network (Figure 5-B.3). Export policies can be specified for every attribute/value pair, and are typically specified by the user. For instance, system administrators will typically restrict ask-delay and ask-veto triggers for shared network devices, such as switches and routers, to only those trusted hosts used for network management and monitoring.

4. APPLICATIONS

We present examples of how the NetQuery programming model enables new functionality based on reasoning about network and client properties.

4.1 Networks checking end hosts

NetQuery enables networks to make decisions based on the properties of end hosts. Network administrators often restrict network access to hosts meeting criteria based on such properties. For instance, our campus network administrators require end hosts to defend themselves and the network against known worms by requiring Windows installations to be up-to-date with patches and to run a virus checker. These requirements are only loosely enforced: compliance is monitored only by a web page where the user checks a checkbox.

NetQuery enables the network to securely query end hosts for the relevant properties. End hosts export tuples describing their operating system, configuration of the network stack, and active processes. Network switches query this information when checking whether a new host is authorized to attach to the network.

Network switches specify an import policy that ensures that this information is genuine. The import policy requires the end hosts to back up their claims with execution certificates, as derived from TPM attestations. These certificates assert that the host is running a operating system that is trusted to make statements concerning the set of running processes. To prevent process-level spoofing, the operating system describes the identifying properties of each process, such as a cryptographic hash of its program image.

Switches initially restrict new hosts to only send and receive packets for NetQuery. When a host first attaches to a switch, it exports its tuples to the local NetQuery server. Upon detecting a new host, a switch checks the host's tuples to determine the list of processes:

$$\texttt{NewHost says } \mathit{NewHost.processes} = [\ \{\mathit{HashLabel} = \texttt{VirusChecker}\},\ \{\mathit{HashLabel} = \texttt{Firefox3.0}\},\ \ldots\ ]$$

If the host runs the required software, then the switch allows it full network access. On doing so, the switch installs a trigger on the process list to detect changes that would indicate a violation of the policy. For instance, the trigger will fire if the virus checker stops running, notifying the switch to revoke the host's network access.

4.2 End hosts checking networks

Service providers often compete solely on price, because the differences between a well-operated, well-provisioned network and a poorly-run competitor are mostly hidden from clients when each offers the same IP forwarding interface. Compared with existing out-of-band mechanisms, exchanging information about differences through NetQuery is more robust and efficient. NetQuery facilitates the use of a service description language that enables service providers to differentiate their service by directly advertising their performance characteristics and empowers clients to exchange information about service providers. Clients can use information self-reported by an ISP or reported by other clients in making service selection decisions.

A high-performance ISP can advertise this fact by describing how its network is operated. For instance, a wireless ISP might perform traffic shaping or QoS to ensure that each end host is allocated a fair share of resources. Access points export statements such as

$$\texttt{AP}_i \texttt{ says } \mathit{wlan0.traffic\_shaping} = \texttt{true}$$

to specify that the features are active on its wireless interface.

To improve efficiency and to hide proprietary details from end hosts, the ISP can use a third party analysis service to assert that its network is well-run. Such services query the tuplespace for an ISP's topology information to determine how it is expected to perform at a particular access point and annotates the ISP with the result:

$\texttt{TPM}_\texttt{i}\texttt{.Linux}_\texttt{j}\texttt{.H323\_Certifier}_\texttt{k}$ `says`
`GoodForVoIP(ISP.HW_MAC#00:00:00:11:11:11)`

Users can also report information about an ISP. For instance, a VoIP application can report a summary of the delivered quality of service:

$\texttt{CiscoATA}_\texttt{i}$ `says GoodForVoIP`$(10.1.2.55)$

Clients use import policies to rely on only trustworthy statements during service selection. By attributing the statements to trustworthy providers or trusted software, NetQuery prevents malicious ISPs or users from injecting inaccurate information. The H.323 certifier is trusted to make accurate statements about a network's ability to support the H.323 telephony protocol: it only bases its analysis on tuples from trustworthy devices. Similarly, Cisco ATA hardware is trusted to make statements based on actual observations.

4.3 Between networks

When two networks are interconnected, network administrators and system designers often concerned about the properties of each network. For instance, ASes generally require their peers to be reliable and behave consistently [2].

Using NetQuery to exchange information between ASes improves security and robustness. NetQuery enables ASes to perform static analysis on the configuration and topology of their peers. If this analysis shows that a peer is properly configured, an AS can accept routes from that peer without adverse performance impacts. For instance, an analysis can use standard techniques to verify that the peer network is robust to single-node failures and propagates only valid routes. Such analyses operate on tuplespace encodings of router configuration files, as expressed in the normalized form from rcc:

`Router1 says` $Router1.sessions = [$
$\{$ $neighbor =$ `10.1.2.3`,
$importPolicy = [$
$\{$ $permit =$ `allow`, $ASregex =$ `^65000`, ... $\}$,
... ...

ASes that enter mutual transit backup relationships can use NetQuery to verify that backup routes are used only when alternate routes are not available. These analyses are implemented using dynamic checking: triggers detect route withdrawals and link failures by monitoring received advertisements and link status:

`EdgeRouter1 says` $EdgeRouter.validAdjRibIn = [$
$\{nextHop =$ `eth0`, $prefix =$ `10.1.2.0/24`,
$ASPath =$ `(22 1 76)`$\}$,
...]
`EdgeRouter1 says` $EdgeRouter.eth0.status =$ `UP`

Changes such as $EdgeRouter1.eth0.status$ transitioning to `DOWN` serve as evidence of failures that can authorize the peer to transmit packets through the backup ISP.

As in preceding examples, the analysis software is supplied by a third party and it executes on a trustworthy platform under the control of the ISP. The verity of the statements contained in the preceding tuples stems from the use of trusted routing platforms. Other import policies can be employed to support legacy routers. For instance, the analysis service can use statements from a trustworthy scraper application that extracts configuration data through router CLI. The scraper only executes on behalf of ISPs that are trusted to operate routers that respond correctly to the scraper's CLI queries.

4.4 Extending functionality with triggers

NetQuery triggers allow the interposition of arbitrary code on network operations. By combining triggers with analysis of end host information, NetQuery clients can extend the network with new functionality without modification to the underlying NetQuery devices.

For instance, a NetQuery switch can allow NetQuery clients to manage access to the network by providing ask-veto access to port status changes. This approach provides more flexibility than embedding a policy within the device (Section 4.1). `NACAgent`, a NetQuery application that enforces a process list-based access policy, uses the trigger

`CHANGE(Switch.ports[*].status)`

to detect when a port connects to a new host. Upon detecting a new host, the switch proposes the change:

`Switch says`
`PROPOSE_CHANGE(`$Switch.ports[0].status =$ `UP)`

The switch does not fully activate the port until after the tuplespace accepts the proposed update, which occurs after all matching triggers are delivered and acknowledged. Upon receiving the trigger, `NACAgent` analyzes the new host's tuples to determine whether it is running an approved set of processes and accordingly signals acceptance or veto in the update's acknowledgment. The host is subsequently allowed to send packets to the network.

A network administrator who wants to restrict potential attacks through unsecured wireless links can modify `NACAgent` to verify that all hosts have deactivated their wireless NICs before connecting to the network. This $\texttt{NACAgent}'$ inspects the host's tuplespace description of its network stack:

`NewHost says` $NewHost.wifi0.status =$ `DISABLED`

NACAgent′ allows access once it verifies that all wireless cards are disabled. As with the process list policy, NACAgent′ adds triggers on the status attribute/value pairs to detect future enablement of the wireless card; upon receiving such changes, NACAgent′ revokes the host's access to the network.

5. IMPLEMENTATION

We have implemented NetQuery on real network hardware and software. The core functionality is composed of a stand-alone tuplespace server and a client library for writing NetQuery devices and applications. We have implemented a NetQuery switch as a Linux user process, a NetQuery router adapted from the Quagga router, a NetQuery host that runs Nexus and exports attested information about its processes and network stack, and applications that rely on the exported properties from these devices.

The tuplespace server, while largely unoptimized, is highly scalable. Since the tuplespace stores representation of network state that already exists elsewhere in the network, albeit not in a format that enables trustworthy dissemination and analysis, the tuplespace server does not need to provide durability. In the event of server failure, the original copy of the network state resides within network devices; to recover from failure, the tuplespace servers instruct the devices to store anew its full tuplespace representation. Thus, a simple in-memory implementation suffices and can provide high performance without compromising correctness. The server implements the distributed tuplespace access protocol, allowing us to manage the load from large test networks by spreading it across multiple tuplespace servers.

The tuplespace server supports the complete set of attribute types and operations. The tuplespace supports a basic trigger predicate and query language. We are extending the predicate, query, and indexing functionality as needed by our applications. Currently, patterns are specified by exact or wildcard match on a single tuple ID and attribute name. Exact match triggers are used during analysis to detect concurrent changes that might invalidate the analysis. For convenience of programming, applications can configure the library to automatically generate such triggers during all reads. We intend to use wildcard matches to build rendezvous mechanism for discovering network devices and invoking network services.

Our work on applications and devices has highlighted the importance of several optimizations. Special indexes for widely-used network state are essential to achieving good performance. For instance, efficiently exporting and exploring the IP level connectivity of a network requires good update and lookup performance for forwarding tables. To achieve this in a compact representation, we imported the Linux implementation of LPC-trie. Although NetQuery, by design, enables clients to establish trust without the assistance of network administrators, trusted tuplespace servers can significantly reduce computational overhead. By trusting a tuplespace server to preserve the integrity of stored statements and to accurately report the speaker of a statement, a NetQuery client does not need to generate and verify digital certificates on every statement exported to and imported from the tuplespace.

5.1 NetQuery devices

The NetQuery switch provides features that enable the derivation of strong network guarantees. The switch exports its internal state as attribute/value pairs that support ask-veto triggers. This provides applications with a great degree of control over network topology, including what devices are attached. The NetQuery switch provides similar functionality to wireless 802.1x to preserve the accuracy of the reported topology. Peer devices are authenticated before they are allowed to connect to the network. Once devices are connected, transparent encryption can be activated to preserve link-layer integrity. These mechanisms enable NetQuery clients to trust that adversaries cannot compromise the accuracy of the tuplespace by subverting the underlying link layer assumptions.

The NetQuery router is Linux-based and derived from Quagga. Only localized changes to the control plane were necessary to convert all interface and routing table changes into NetQuery attribute/value pairs that support ask-delay triggers. NetQuery-Quagga interposes on Quagga's interface to rtnetlink, Linux's low-level interface to its in-kernel dataplane, and exports all relevant requests, such as changes to the forwarding table and NIC state, to the tuplespace. In total, only 777 lines of localized changes were needed, out of a total code base of 190,538 LOC.

The NetQuery switch and router export connectivity information about the NetQuery-enabled peers that are attached to their local interfaces. Exporting connectivity information enables applications to explore the network by crawling its tuple representation. Unlike the NetQuery-Switch, NetQuery-Quagga does not support veto triggers; doing so would have required extensive changes to Quagga.

5.2 NetQuery applications

We have built several applications that use the tuplespace to extend the functionality of these devices and to provide applications with guarantees about the network. We used triggers to extend the NetQuery switch with a custom network access control (NAC) policy, described in Section 4.4, that allows access only to Nexus hosts that are running virus checkers. We implemented a generic route characterization analysis, which determines the route between a source and destination, and detects any changes that can modify the route. This analysis is a building block for more complex analyses of route characteristics; for instance, an application can check whether a route respects user privacy by checking that none of the entities on the route perform packet inspection.

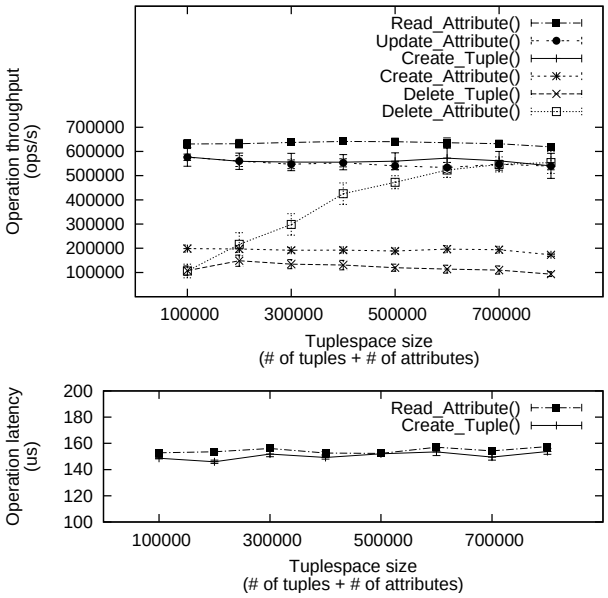

Figure 6: **NetQuery operation throughput and latency versus tuplespace size.** The performance of all operations is independent of the tuplespace size.

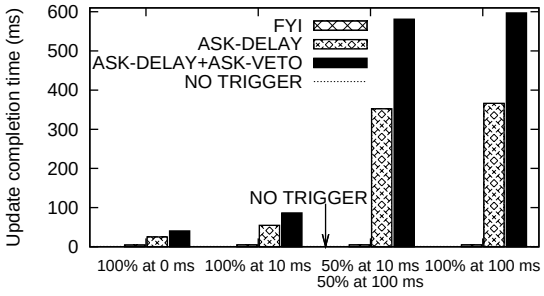

Figure 7: **Trigger-induced update completion time overhead.** The types of trigger and maximum server to client RTT, rather than the number of triggers, dominate trigger overhead.

6. EVALUATION

NetQuery performs well for both ISPs and applications: it can be integrated into network devices with little overhead, efficiently deployed on large networks and provide strong guarantees and good throughput for applications. All experiments used a testbed consisting of Linux 2.6.23 hosts equipped with 8-core 2.5GHz Intel Xeon processors and connected over a Gigabit Ethernet switch. Cryptography is disabled in all experiments.

6.1 Microbenchmarks

We show through microbenchmarks that NetQuery can support large tuplestore and immensely high tuple creation, modification, and deletion rates. Thus, the unoptimized prototype is capable of supporting the peak needs of an ISP. The microbenchmark was distributed across sixteen NetQuery client processes, running on a single machine with eight cores, and eight tuplespace server processes, running on a single, separate machine with eight cores. In the throughput experiment, the microbenchmark issued a sequence of pipelined requests, without waiting for responses from the server until all were issued (Figures 6). No forwarding pointers were followed for these operations, since every attribute/value pair resided on the same server as its tuple.

With the exception of **Delete_Attribute**(), the performance of all tuplespace operations is decoupled from the size of the tuplespace. For smaller tuplespace sizes, the **Delete_Attribute**() experiments were of such short duration that initialization overheads dominated the overall execution time. The reduced throughput of **Delete_Tuple**() was due to the initial tuplespace state for this experiment, wherein every tuple was initialized with nine attribute/value pairs. Deleting a tuple also entailed deleting its attribute/value pairs.

The latency experiments used one server and client and only measured operations where the client necessarily waits for data. For instance, **Create_Tuple**() and **Read_Attribute**() returned the tuple ID of the new tuple and attribute value, respectively.

The latency of a tuplespace operation can also depend on any induced trigger notifications; such operations may require the server to wait for a response from a trigger recipient. Sending trigger notifications in parallel limits such impacts. Figure 7 shows the trigger-induced delays to an attribute update under trigger workloads that varied by trigger type and WAN delay between the tuplespace server and trigger recipients. RTT and trigger types were the primary determinants for delay. Since trigger notifications were sent in parallel, the overall completion time for the update time was dominated by the latency of the most expensive notification, rather than the total number of trigger notifications. This experiment shows that NetQuery can support a large number of WAN end hosts that use FYI triggers. While ask-delay and ask-veto is less scalable, these are intended to support management and control applications, which typically run on a small number of hosts.

6.2 Performance on real-world traces

To demonstrate that NetQuery has low device overhead and efficiently utilizes server resources, we measured the NetQuery overheads associated with reflecting device state changes to the tuplespace and the corresponding overhead for tuplespace servers. Our exemplar device was a NetQuery Quagga BGP router and was subjected to a realistic workload, derived from a RouteViews trace. The Quagga router and tuplespace servers were executed on separate machines. The workload generator converted the RouteViews data into a BGP stream and ran on the same machine as the router.

To demonstrate that NetQuery can efficiently import bulk data from NetQuery devices, we measured the completion time for router initialization and the tuplespace memory footprint of a router. We used RouteViews RIB traces to generate a full BGP routing table transfer (268K prefixes) that fully initialized a router's forwarding table; such transfers were exceptionally intensive for both the router and tuplespace server. Upon receiving the transfer, the BGP router downloads a full forwarding table to the IP forwarding layer. The standard BGP router completed in 5.70 s, while the NetQuery BGP router completed in 13.5 s.

To demonstrate that NetQuery routers perform well in steady state, we evaluated the router against a workload derived from RouteViews update traces. The workload generator sent updates in 1 second batches. The experiment recorded the time needed to completely commit the resulting changes to the IP forwarding tables and to the tuplespace. We measured NetQuery's performance on a total of 30 hours of update traces, composed of three hour traces from 10 randomly selected source routers.

The standard Quagga server had a median completion time of 62.2 ms, while the NetQuery Quagga server had a median completion time of 63.4 ms. Thus, the NetQuery router can react to network changes almost as quickly as a standard router, minimizing the disruption to local forwarding.

The overhead introduced by NetQuery does not increase convergence time even under conservative router implementation assumptions. Suppose the overhead is in the critical path of BGP update propagation, either because RIB information is to be exported, or because the FIB update blocks BGP update propagation. Since BGP convergence is governed by mandatory artificial delays to BGP update propagation, NetQuery's overheads do not impact convergence time. For instance, BGP's MRAI timer delays the propagation of all BGP updates by 30 seconds, sufficiently long to hide NetQuery's initialization and steady state overheads. Even the initialization overhead, the largest possible forwarding table change, is shorter.

The full 268K entry forwarding table consumed only 10.7 MB of server memory. The unoptimized NetQuery wire protocol required 62.8MB to transmit the full table at initialization time. NetQuery required only 3 KB, 92 KB, and 480 KB to transmit the median, mean, and maximum update sizes seen during steady state.

Performance within ISP networks.

Here, we show through analysis and experiments that a small number of NetQuery servers can collect tuplespace updates from every router in a POP without degrading performance.

We can see this by using the results from the previous experiment to estimate the necessary resources to deploy NetQuery. The largest POP in the Sprint RocketFuel topology consists of 66 routers [19]. Storing the full forwarding tables of all routers required 0.66 GB of memory, easily fitting within the 16 GB memory capacity of our test machine. Since tuplespace servers are local to the POP, the burst of traffic during initialization traverses high capacity local links. Assuming a conservative lower bound of 1 s between routing updates, the median bandwidth utilization is less than 198 KB/s, or less than 0.2% of a gigabit link.

To determine the number of servers needed to process tuplespace updates without delaying BGP convergence, consider the processing bottlenecks. The MRAI timer sets a constraint of 30 s on completion time. In this worst case, all routers are reloading at the same time, ensuring that load-balancing is close to ideal. For 66 routers and initialization overhead of 13.5 s, 30 cores, or four machines, are needed to meet this deadline. These constraints can be relaxed if the ISP and users are willing to make tradeoffs during periods of high network disruption. For instance, high availability applications might forgo guarantees in exchange for minimizing downtime, while security conscious applications might accept increased downtime in exchange for preserving guarantees.

To validate these results, we ran an experiment to measure the aggregate route update overhead induced by routers within a real ISP topology. We used the Sprint RocketFuel topology, which consists of 17,163 Sprint and customer edge routers. Although the resulting forwarding table are smaller, IGP deployments can have stricter deadline constraints: since routing instability is easier to manage within a single AS, routers are often configured to propagate updates as quickly as possible to minimize convergence time. The workload consisted of forwarding table updates for internal routes, generated through simulation of hierarchical shortest path routing. The resulting trace for a single POP was fed to a real tuplespace server over a Gigabit Ethernet link.

We measured for each POP the completion time of initialization and of updates resulting from a set of correlated link failures, with the failure rate varying between 0.01 and 0.09. For the five largest POPs, each consisting of 51 to 66 routers, the median initialization time for loading all routing table entries for every router was 13.3 s. For failure rates of up to 0.05, the mean and median update times of the POP were less than 0.24 s and 0.14 s, respectively; these local delays to forwarding table updates are negligible in most networks. This delay does not impact global routing convergence for typical link-state routing protocols such as OSPF. Such delays typically result if flooding is delayed. In general, routing protocol implementations decouple flooding from RIB/FIB calculations. Since NetQuery only hooks onto the critical path of forwarding table updates, no delay is introduced to the critical path for flooding.

6.3 Macrobenchmarks

To determine the cost of using NetQuery to establish guarantees, we measured the performance of the route characterization analysis, which is used by several applications to establish path guarantees. We ran the query on a set of Net-

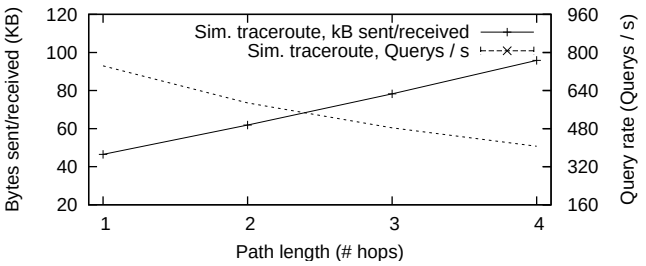

Figure 8: **Route characterization analysis.** The network and processing overhead increase linearly with path length, as does completion time.

Query Quagga servers arranged in a line topology and connected to a single tuplespace server process. The query cost and execution time increases linearly with path length (Figure 8). Route characterizations enables applications to determine path properties in a trustworthy fashion. Although standard traceroute is less expensive than route characterization, it is not capable of supporting such applications.

7. RELATED WORK

Network design has shifted towards using a logically centralized server to make control decisions on behalf of every device in a network, rather than distributing control across multiple devices [5, 9, 3]. Similarly, NetQuery centrally collects network properties to isolate devices from the performance impact of queries and triggers.

Perhaps the closest work to NetQuery are network exception handlers (NEH) and NOX, which are logically centralized systems for enterprise networks that collect network properties and disseminate it to administrative applications [14, 10]. NEH collects dynamic topology, load statistics, and link costs from the network infrastructure, and exposes these properties so that end hosts can detect and react to exceptional network conditions. NOX collects information about topology from infrastructure devices and user/service bindings concerning end hosts and exposes this information to network management applications running on a central controller. NetQuery supports these applications for enterprise networks, while also supporting applications that issue queries that span multiple ASes. NetQuery's tuplespace supports heterogeneous information sources by tracking the source of every statement and by leveraging trusted hardware. While NEH and NOX are intended for applications specified by the network administrator, NetQuery enables any user application to query the network.

Secure Network Datalog (SeNDlog) is a declarative programming system for building distributed applications on untrusted networks [22]. With declarative programming systems, distributed applications are written as high-level rules, which are compiled to query execution plans that are executed at every node. SeNDlog extends declarative programming with logic-based access control. As with NetQuery, SeNDlog uses `says` as the basis for decentralized authorization policies.

Network management systems are widely deployed on the Internet. Intradomain management protocols such as SNMP and CMIS/CMIP provide standard schemas and protocols for disseminating information about network devices [11, 13]. NetQuery can be retrofitted on top of existing SNMP and CMIS/CMIP interfaces. RPSL is the standard language for exporting information about ASes across administrative boundaries [1]. This information is disseminated through the Internet Routing Registry (IRR), a globally distributed database for RPSL statements. Although network analyses have been designed using this system [18], the questionable accuracy of IRR information, which stems from inconsistent update mechanisms across the ISPs that contribute information, limits the guarantees that applications can derive from such analysis. NetQuery's rich policy language enables applications to reason about the verity of tuplespace information.

Trusted Network Connect (TNC) is an emerging standard for network access control of end hosts [21]. Client authorization in TNC is similar to that of NetQuery. A network administrator specifies a network access policy to restrict access to the network. Before an end host is allowed to join the network, TNC verifies that its software and hardware configuration satisfies the policy. Unlike NetQuery, TNC does not provide a channel by which the end host can discover the properties of the network before deciding to connect.

Internet measurements are used to derive properties of the network such as network topology or predicted network performance [19, 20, 16]. In addition to providing information for these inference tools, NetQuery can serve as a medium for network information service providers to publish the results.

8. CONCLUSION

NetQuery provides a new channel for disseminating the properties of network entities. The architecture of the system is well-suited to the scale of the Internet and supports the hardware provisioning, confidentiality policies, and trust relationships found in heterogeneous networks. It is administratively decentralized, provides a flexible tuple-based data model, and supports safe analysis. These features reduce the barrier to entry for new kinds of network information services and applications, and facilitates incremental deployment in legacy networks. We have experimentally shown that NetQuery performs well on real routers, and can support the volume of network state found in large enterprise and ISP networks. NetQuery provides a general programming model that appropriate for a diverse range of applications, including verifying contractual obligations, enforcing user-specified policies, and enhancing performance.

NetQuery is well-positioned to leverage the trusted hardware that is increasingly available within networks. We believe that a general-purpose channel for disseminating se-

cure statements issued by trusted hardware as well as signed statements from network operators and users will enable a new and exciting class of applications based on meta-level reasoning about the state of the network.

9. REFERENCES

[1] C. Alaettinoglu, C. Villamizar, E. Gerich, and D. Kessens. RFC 2622: A Framework for IP Based Virtual Private Networks. 1999.

[2] AOL. AOL Transit Data Network: Settlement-Free Interconnection Policy, 2006. http://www.atdn.net/settlement%5Ffree%5Fint.shtml.

[3] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and Implementation of a Routing Control Platform. In *USENIX/ACM NSDI*, May 2005.

[4] M. Casado, P. Cao, A. Akella, and N. Provos. Flow-Cookies: Using Bandwidth Amplification to Defend Against DDoS Flooding Attacks. In *IEEE IWQoS*, June 2006.

[5] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. *SIGCOMM CCR*, 37(4):1–12, 2007.

[6] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A Trusted Open Platform. *Computer*, 36(7):55–62, 2003.

[7] N. Feamster and H. Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *USENIX/ACM NSDI*, May 2005.

[8] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The Digital Distributed System Security Architecture. In *Proc. 12th NIST-NCSC National Computer Security Conference*, pages 305–319, 1989.

[9] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A Clean Slate 4D Approach to Network Control and Management. *SIGCOMM CCR*, 35(5):41–54, 2005.

[10] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. *SIGCOMM CCR*, 38(3):105–110, 2008.

[11] D. Harrington, R. Presuhn, and B. Wijnen. RFC 3411: An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks. Dec. 2002.

[12] S. Incorporated. Meeting the Challenge of Today's Evasive P2P Traffic: Service Provider Strategies for Managing P2P Filesharing, 2004. http://www.sandvine .com/solutions/pdf/Evasive%5F P2P%5FTraffic.pdf.

[13] International Organization for Standardization. ISO DIS 9596-2: Information Processing Systems - Open Systems Interconnection, Management Information Protocol Specification - Part 2: Common Management Information Protocol. Dec. 1988.

[14] T. Karagiannis, R. Mortier, and A. Rowstron. Network Exception Handlers: Host-network Control in Enterprise Networks. In *ACM SIGCOMM*, Aug. 2008.

[15] T. Karygiannis and L. Owens. Wireless Network Security: 802.11, Bluetooth and Handheld Devices. Special Publication 800-48, NIST, November 2002.

[16] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An Information Plane for Distributed Services. In *USENIX/ACM OSDI*, Nov. 2006.

[17] NebuAd. Juniper Networks partners with NebuAd to enable ISPs to participate in online advertising revenues on the web. http://www.juniperamspmarketing.com/NebuAD.htm.

[18] G. Siganos and M. Faloutsos. Analyzing BGP Policies: Methodology and Tool. In *IEEE INFOCOMM*, Mar. 2004.

[19] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. Measuring ISP topologies with Rocketfuel. *IEEE/ACM Transactions on Networking*, 12(1):2–16, 2004.

[20] N. Spring, D. Wetherall, and T. Anderson. Reverse Engineering the Internet. *SIGCOMM CCR*, 34(1):3–8, 2004.

[21] Trusted Computing Group. *TCG Trusted Network Connect: TNC Architecture for Interoperability, Specification Version 1.3*. Trusted Computing Group, Apr. 2008.

[22] W. Zhou, Y. Mao, B. T. Loo, and M. Abadi. Unified Declarative Platform for Secure Networked Information Systems. In *IEEE ICDE*, Apr. 2009.