An Axiomatic Basis for Communication

Martin Karsten[1], S. Keshav[1], Sanjiva Prasad[2], Mirza Beg[1]
[1] David R. Cheriton School of Computer Science, University of Waterloo
{mkarsten,keshav,mbeg}@cs.uwaterloo.ca
[2] Department of Computer Science and Engineering, IIT Delhi
sanjiva@cse.iitd.ac.in

ABSTRACT

The de facto service architecture of today's communication networks, in particular the Internet, is heterogeneous, complex, ad hoc, and not particularly well understood. With layering as the only means for functional abstraction, and even this violated by middle-boxes, the diversity of current technologies can barely be expressed, let alone analyzed. As a first step to remedying this problem, we present an axiomatic formulation of fundamental forwarding mechanisms in communication networks. This formulation allows us to express precisely and abstractly the concepts of *naming* and *addressing* and to specify a consistent set of control patterns and operational primitives, from which a variety of communication services can be composed. Importantly, this framework can be used to (1) formally analyze network protocols based on structural properties, and also to (2) derive working prototype implementations of these protocols. The prototype is implemented as a *universal forwarding engine*, a general framework and runtime environment based on the Click router.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Store and forward networks

General Terms

Design, Languages, Theory, Verification

Keywords

Concepts, Definitions, Naming, Addressing, Routing, Protocols

1. INTRODUCTION

Traditionally, the Internet is modelled as a graph, where each node implements a set of protocol layers and each edge corresponds to a physical communication link. Unfortunately, when compared with the actual Internet, this model falls short. In the traditional model, nodes are addressed by one or more static IP addresses. End systems implement a simple five-layer stack, with applications using a transport layer to access IP, which is layered on the data link and physical layers. Packet forwarding decisions are made purely on the basis of IP 'routing' tables. Moreover, a protocol layer at any node only inspects packet headers associated with that layer, obeying strict rules in dealing with other layers. In reality:

- DHCP, anycast, multicast, NAT, mobile IP and others break the static association between a node and its IP address.
- Nodes implement more layers, including IP or VLAN tunnels, overlays, and shims, such as MPLS.
- Forwarding decisions are made not only by IP routers, but also by VLAN switches, MPLS routers, NAT boxes, firewalls, and wireless mesh routing nodes.
- Middleboxes and cross-layered nodes such as NATs, firewalls, and load balancers violate layering.

In face of these significant extensions to the classical model, understanding the topology of the Internet in terms of its connectivity has become a daunting task. It has become difficult to define elementary concepts such as a neighbour and peer relationships, let alone the more complex processes of forwarding and routing. Further, there is not even a common and well-defined language for fundamental networking concepts, with terms such as 'name', 'address', or 'port' being the subject of seemingly endless debate.

Yet, surprisingly, the system still works! Most users, most of the time, are able to use the Internet. What lies behind the unreasonable effectiveness of the Internet? We postulate that all extensions to the traditional model, no matter how ad hoc, obey a set of underlying principles, which preserve connectivity. However, these principles have rarely been systematically studied (with [5, 8] being notable exceptions).

Our research goal is to axiomatically specify basic internetworking concepts that allow us to construct (a) a theoretically sound framework to express architectural invariants – such as the deliverability of messages – even in the presence of network dynamism, middleboxes, and a variety of compositions of different protocols, (b) an expressive meta-language in which to rapidly implement a variety of packet forwarding schemes, and (c) an integrated model that correctly describes packet progress across multiple layers of communication protocols. The concepts and the meta-language derived from them serve not only to clarify the essential architecture of the Internet, but also provide a bridge between formal proofs on node reachability using a particular forwarding scheme and a practical implementation of that scheme. Our goals are inspired by Hoare's axiomatic basis for programming [10]. We believe that the conceptual clarity that arises from our work allows us to quickly sketch the essential aspects of any type of communication network,

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

no matter how exotic, and apply concepts from one network technology to another.

To keep the problem tractable, we propose to split overall communication functionality into two broad areas: one area is concerned with *connectivity*, i.e., naming, addressing, forwarding, and routing. The second is the set of mechanisms to provide additional functionality related to communication quality and performance. This includes medium access control, reliability, flow control, congestion control, security, among others, and is not yet explicitly considered in this work. In particular, the framework presented here is oblivious to time and cannot model loss and timeouts.

The paper is organized as follows. After presenting related work in Section 2, we introduce the axiomatic framework for message forwarding in Section 3 and add control considerations in Section 4. Section 5 demonstrates the generality of this framework by providing uniform pseudo-code for some forwarding schemes. Section 6 explores the semantic foundations with a proof system supporting formal verification. Section 7 illustrates the practical capabilities of our approach by compactly describing seemingly diverse networking techniques such as TCP over NAT, Hierarchical Mobile IP, and I3. Section 8 outlines a prototype implementation based on Click and the paper is concluded with a discussion in Section 9.

2. RELATED WORK

Our work draws from and is related to a handful of other attempts to bring clarity to Internet architecture. Clark's seminal paper [5] succinctly lays out the design principles of the classical Internet, but does not provide a basis for formal reasoning about its properties. Recently, Griffin and Sobrinho have used formal semantics to model routing [8] and Loo et al. have used a declarative approach to describe routing protocols [16]. Our work differs in that we focus on the elementary notions of forwarding, naming, and addressing.

Our work is directly related to past work in the area of naming and addressing indirection. This has been considered both in existing technology standards, such as IP Multicast, IPv6, or Mobile IP, as well as in recent research proposals [3, 9, 17, 21]. Similar to our work, these proposals blur the traditional distinction between naming and addressing, and also consider innovative packet forwarding mechanisms. However, to our knowledge, these past proposals are essentially ad hoc, without a consistent set of underlying formal principles. In contrast, we suggest an axiomatic formulation of communication principles and thereby present a first attempt at building a complete formal basis for reasoning about communication systems. In earlier work, we have made an attempt at an axiomatic formulation of communication principles [13]. The present work is significantly more comprehensive, clearly states the axiomatic basis, and the formalization is based on high-level Hoare-style assertions, rather than low-level operational semantics.

Novel architectures for naming and addressing have been proposed for a new generation of 'pocket-switched' [20], 'ambient' [2] and 'delay-tolerant' [6] networks. These new architectures are in response to fundamentally different networking paradigms. We believe that our formalizations are adequate to represent these non-traditional naming and forwarding architectures.

In the past, other authors have also attempted to generalize Internet concepts, recognizing the failure of the classical model to adequately describe ground realities and conceptual isomorphisms. The Multi-Domain Communication Model [24] is an example of such a generalization. However, this, and similar generalizations, do not have an axiomatic foundation, and therefore tend to be ad hoc. Ahlgren et al. [1] have previously suggested that the Internet architecture has been guided by some invariants. We agree with their viewpoint: our contribution is to formalize these invariants.

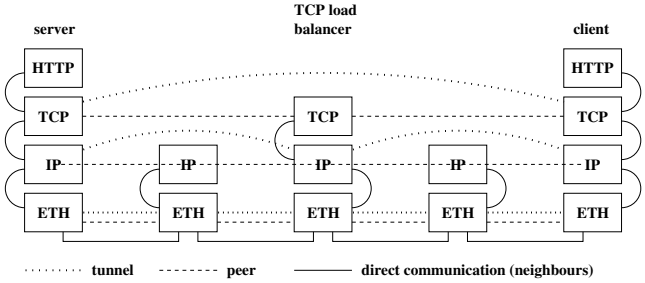

Figure 1: Protocol Layers

In recent work, Zave [25] has used declarative semantics to describe and validate several alternative naming and binding schemes, especially as they relate to 'returnability' of a call setup message. We believe that this approach is complementary to ours.

Finally, several research projects have come up with 'protocol engines' that can be used to rapidly implement complex protocol architectures. These include the x-Kernel [11], micro-protocol composition [15], and Click [14] that we ourselves build upon. Although these systems ease the code development process, and the micro-protocol approach uses NuPrl to provide proofs of correctness, they do not develop an axiomatic foundation. Moreover, they do not deal with multi-hop and multi-layer forwarding and name resolution.

3. AXIOMATIC FRAMEWORK

3.1 Definitions

Before presenting an axiomatic formulation of forwarding principles, we define a few concepts.

- An *abstract switching element (ASE)* is an object that participates in network communication and relays messages. It generalizes a simple switching element, such as the table-based crossbar switch in Autonet [19], to an abstract object that is more representative of a protocol layer in a general communication network. That is, in addition to switching messages, an ASE can carry out more complex actions, such as swapping header labels and encapsulating a message into another.

- An ASE has named input and output communication ports. These correspond to physical network interface cards or *logical ports*. Simplex *direct communication* from the output port of an ASE to the input port of an adjacent ASE is accomplished by means of shared memory or a physical medium such as cable, radio, or fibre. At ASE B, the input port from a *predecessor* ASE A is denoted as ${}^{A}B$ and the output port to a *successor* ASE C is B^{C}. We use a lower case superscript such as ${}^{x}B$ or B^{x} to express the previous or next ASE as variable x. Note that different logical ports may refer to the same physical port. Referring to Figure 1, where each rectangle represents an ASE, examples of direct communication are between the TCP and IP ASEs on the same machine and two Ethernet ASEs on the same shared medium.

- The unit of communication is a *message*, which is a string of identifiers drawn from an arbitrary alphabet. In the Internet, a message corresponds to an application-level data unit, datagram, or MAC-frame. A message m that exists at a port x is denoted as $m@x$.

- We denote with ${}^0B$ the logical port that is equivalent to *creating* a message at ASE B and B^0 the logical port that *consumes* a message. Creation and consumption refer to a transformation of the message in or out of the realm of this theoretical model. In Figure 1, the HTTP ASEs create and consume application-level messages, and the TCP ASEs create and consume acknowledgement messages.

- An ASE maintains a private set of mappings, called its local *switching table*. The switching table at ASE B is denoted as S_B and contains mappings $\langle A,p\rangle \mapsto \{\langle C,p'\rangle\}$ from a ASE-and-string pair $\langle A,p\rangle$ to a **set** of ASE-and-string pairs $\langle C,p'\rangle$. For notational simplicity, we represent an ASE X in local mappings as X. The switching table can be queried through an exact lookup operation $S_B[A,p]$. If no exact mapping exists in the switching table, the message is discarded. Examples of a switching table are Ethernet or IP forwarding tables with $p' = p$ (ignoring the TTL decrement operation, for now).

3.2 The Axioms

The "leads-to" relation provides the central axiomatic formulation of operations of store-and-forward networks.[1] The "leads-to" relation is denoted as $\rightarrow$ and defined by the following four axioms:

LT1. (Direct Communication)
$\forall A,B,m : \exists A^B, {}^AB \iff m@A^B \rightarrow m@{}^AB.$

LT2. (Local Switching)
$\forall A,B,C,m,p,p' : \exists {}^AB, B^C \wedge \langle C,p'\rangle \in S_B[A,p]$
$\implies pm@{}^AB \rightarrow p'm@B^C.$

LT3. (Transitivity)
$\forall x,y,z,m,m',m'' : (m@x \rightarrow m'@y) \wedge (m'@y \rightarrow m''@z)$
$\implies m@x \rightarrow m''@z.$

LT4. (Reflexivity) $\forall m,x : m@x \rightarrow m@x$

Axiom LT1 describes direct communication between ASEs. A^B and ${}^BA$ exist if and only if A can directly communicate with B. Axiom LT2 expresses the lookup and switching capability of an ASE. Note that a message pm is logically split into a header prefix p and the opaque rest of the message m during each local switching step. LT2 also covers any form of multi-recipient forwarding, such as multicast, since $S_B[A,b]$ may have multiple elements. Axiom LT3 splices individual forwarding steps together. These three axioms naturally capture the simplex forwarding process in a communication network, where, potentially, at each forwarding step, a forwarding label is swapped. To simplify formal proofs of certain reachability properties, reflexivity is axiom LT4.

To illustrate the applicability of this model, consider the special case of LT2 where $p = p'$. In this case, the "leads-to" relation describes a single-layer forwarding system based on global destination addresses, such as IP forwarding. As another special case, consider the absence of prefixes altogether. Then, the axioms describe a forwarding model based on input and output ports, such as circuit switching. In general, as we demonstrate later, we claim that the "leads-to" relation can describe arbitrarily complex multi-layer forwarding systems.

In a real network, network messages typically contain a stack of protocol headers that carry different types of names, such as 'addresses', 'protocol numbers', or other identifiers. The "leads-to" relation as defined on an arbitrary message string is just an abstract representation of these forwarding mechanisms. Typically, the destination identifiers (and sometimes the source identifiers) in each respective protocol header constitute the prefix p. However, the conceptual model works with an arbitrary subset of protocol header fields. Some non-trivial examples are presented in Section 7.

[1] It is inspired by Lamport's "happened-before" relation.

3.3 Communication Concepts

Based on the "leads-to" relationship, we can succinctly define and explain a number of well-known communication concepts, beginning with a formal definition of a name:

Name If $\exists$ ASEs A,B and prefix $p \neq \varnothing$ such that $\forall m$:
$pm@^xA \rightarrow p'm@^yB \rightarrow m@B^z$ and $p' \neq \varnothing$,
then p is a name for B at A.

The name of an ASE is the prefix that is removed when the message is transmitted to this ASE. Note that p can be a name at A for multiple ASEs. The condition $p' \neq \varnothing$ ensures that B is indeed the ASE where the prefix or any residual of it is removed. Any string that "leads to" a particular ASE B from origin ASE A is considered a name for B at A. By default, names are local and relative to the ASE from which they originate. For example, a message header could contain a stack of labels, each of which identifies forwarding state at subsequent ASEs (also known as *source routing*) along a particular path from a source ASE to a destination ASE. The complete stack of labels would then be a name for the destination ASE relative to the source ASE.

There is no pre-existing formal and universally accepted definition for "name" in communication networks and the above definition may not match everyone's intuition. Our model at least enables a formal yet intuitive definition in the first place, which should add some clarity to the debate.

Address If $\exists$ ASEs A,B and prefix $p \neq \varnothing$ such that $\forall m$:
$pm@^xA \rightarrow pm@^yB \rightarrow m@B^z$,
then p is an address for B at A.

We define an address as a special kind of name that does not change along the path. In other words, if an ASE writes a name into a message with the assumption that at least some other ASEs interpret the string to send it to the same destination, it becomes an address. Every address is also a name, but the reverse is not true. Note that p can be an address for multiple ASEs.

Peer If $\exists$ ASEs A,B and prefixes p,p' such that $\forall m$:
$pm@A^x \rightarrow p'm@^yB$ and $p' \neq \varnothing$ and
$S_A[x,p] \neq \varnothing$ and $S_B[y,p'] \neq \varnothing$, then A and B are *peers*.

Tunnel If $\exists$ ASEs A,B and prefixes p,p' such that $\forall m$:
$m@^wA \rightarrow pm@A^x \rightarrow p'm@^yB \rightarrow m@B^z$ and $p' \neq \varnothing$,
then A and B form a *tunnel*.

The difference between direct neighbours, peer, and tunnel is illustrated in Figure 1. Peer ASEs operate on an identical portion of the header prefix, which typically corresponds to the same protocol header field(s). Note that tunnels are between two peers, but may traverse additional peers. For example, an IP sender, IP routers, and an IP destination are peers, but not direct neighbours because (a) they all operate on the same IP header, and maintain a local switching table indexed by the IP destination address found in the IP header and (b) IP ASEs never directly communicate with each other; they communicate through a link-layer ASE. The IP sender and IP destination form a tunnel, because the definition of tunnel is satisfied for m = the IP payload, p = the IP header. A pair of connected TCP endpoints also form a tunnel for similar reasons. A

pair of connected Ethernet NICs can be considered as both direct neighbours and peers. Note that tunnels are similar to ISO protocol interfaces, whereas direct communication occurs over service interfaces.

Name Scope The scope of a name p for a set of ASEs β is denoted as σ_p and defined as follows:
$\exists\beta : \forall$ ASEs $A_i : p$ is a name for β at $A_i \Longrightarrow A_i \in \sigma_p$

Message Scope The scope ρ_m of a message $m@^y x$ is defined as the scope of the outermost destination name:
$m@^y x = pl@^y x \wedge \exists S_x[y,p] \Longrightarrow \rho_m = \sigma_p$.

If a name scope encompasses only one ASE, we call it *local*. A set of names or addresses with the same scope is called *name space* or *address space* respectively.

3.4 Naming and Binding Revisited

We now take a detour to compare our framework and definitions with Saltzer's seminal work on naming, addressing, and binding [18]. Saltzer's work did not consider communication in distributed systems; nevertheless, his insights are consistent with the axiomatic framework resulting from the "leads-to" relation.

In Saltzer's model, a *name* is a string that is used to refer to an object "in the system". We precisely define the name of an ASE to be the string that can be used to reach it. Moreover, the concept of a name is extended to refer to a set of objects (to support broadcast, anycast, or multicast communication).

Saltzer uses the term *binding* to refer to the establishment and existence of a relation between a name and an object. In our framework, the existence of a *local binding* (i.e. local switching state) can be stated precisely as $\exists(\langle A,p\rangle \mapsto \langle B,p'\rangle) \in S_A$. In addition, we introduce the additional concept of *distributed binding* that is established by the chain of local switching states, such that a name "leads to" a set of objects that have a local binding for this name. We can then define *distributed resolution* as forwarding a resolution request and getting back the appropriate response. That is, distributed resolution can be considered as forwarding a resolution request message using the requested name as destination address. The forwarding of resolution requests is similar to the definition of *closure* in Saltzer's work [18], which is defined as 'the mechanism that connects an object wishing to resolve a name to a particular context'.

Saltzer defines a *context* as a set of bindings. A name is always interpreted relative to some context. Both the local switching table in an ASE, as well as the distributed context formed by a name space can be regarded as instances of this notion.

In summary, we claim that our framework is a natural extension of Saltzer's single-system definitions to a communication network.

3.5 Forwarding Operations

In theory, the transformation from p to p' in LT2 in Section 3.2 is unrestricted, but in practice it is either a *push*, *pop*, *swap*, or *nop* operation, as described next. In case of *push*, a new prefix q is prepended and $p' = qp$. In case of *pop*, p is removed and $p' = \varnothing$. In case of *swap*, p is replaced by p', which usually is of equal length. With *nop*, p remains unchanged and $p' = p$. Given these transformations, it is possible to identify corresponding forwarding operations that cover a wide range of forwarding techniques used in communication protocols.

Nop - Forwarding If no modification of the current name takes place, the message is just forwarded. Ethernet bridging or IP forwarding are prominent examples, but circuit switching also trivially falls in this category.

Push - Encapsulation The push operation stacks a new name "on top" of the existing stack. This is used at tunnel ingress points, but also as a general mechanism for protocol layering, e.g., placing the network addresses in front of the protocol number in front of the transport ports. A special case of push-encapsulation is *route recording* where the name of the current node is added to the protocol header, as in Dynamic Source Routing (DSR) [12].

Pop - Decapsulation The pop operation removes the "topmost" name from the stack. It is used at tunnel egress gateways and the receiving side of layered protocols, as well as when forwarding a source-routed message.

Swap - Label Switching The swap operation replaces the current name. This is used in virtual circuit networks such as MPLS or ATM, but also when forwarding a packet from the external to the internal network at a NAT box. We note that, by introducing a new name, a swap operation typically changes the message scope.

A special consideration applies to the pop operation. When a message is processed at an ASE, the pop operation is carried out only if it is determined that the removed information is indeed no longer needed to sufficiently identify the sender. Otherwise, the information is logically removed (from the viewpoint of the "leads-to" relation), but physically stays with the message. This is a configuration arrangement between ASEs, such that an upstream ASE delays its pop operation depending on which downstream ASE a message is forwarded to. For example, Ethernet source information of a packet carrying both an IP and Ethernet header can typically be removed, since the IP source address is sufficient to return to the sender, but IP source information of an IP packet carrying a TCP segment cannot be removed immediately, since it is needed to demultiplex between different TCP connections. We call this *delayed pop* and describe how it relates to the "completeness" of a name in Section 4.1.

Using the concepts introduced so far, it is possible to describe basic data path mechanisms of a communication network. For example, we can precisely represent a tunnel between two TCP ASEs that is established by a three way handshake. Similarly, a transient HTTP tunnel exists between a browser client and a web server for the duration of the TCP tunnel between them. An HTTP load balancer that examines the HTTP header would be a peer of the browser, and it would also be a peer to the web server. The load balancer is a forwarding engine, just like an IP router.

We can also describe arbitrary static forwarding scenarios, where forwarding tables and topologies do not change over time, and thus static switching tables are sufficient to determine the successor ASEs for forwarding. As an example, consider an IP network with pre-configured routing tables, running over Ethernet with all ARP lookups also pre-configured in the ARP cache, as in Figure 1.

3.6 Forwarding Primitives

The processing operations stated above can be translated into a set of forwarding primitives that can be used to abstractly specify ASE processing in pseudo-code. Each ASE supports the same set of operational primitives and ASEs essentially differ only in the implementation of these standard interfaces.

We first introduce the abstract data types that are necessary to describe the forwarding primitives. The *message* type abstractly refers to the internal representation of a communication message. A *string* represents an arbitrary string that is taken from or placed into a message. In an actual implementation, this would be replaced by an ASE-specific data structure for efficiency. Finally,

the *ase* generic data type represents the ASEs in the system. Note that we use multi-return specifications where applicable to keep the pseudo-code concise. The forwarding primitives below are used in Section 5 to specify the abstract functionality of a network element.

- `send(ase, message)`
  The send operation implements sends a message to the given ASE. Because the pseudo-code does not explicitly represent ports, the ASE is passed as a parameter.
- `<ase, message> receive()`
  This primitive is used to receive messages and returns the message *as well as* the ASE from which the message has been received.
- `message copy(message)`
  It is necessary to copy messages to implement LT2 for mappings in the local switching table that result a set of ASE-and-string pairs.
- `push(message, string)`
  The push operation is ASE-specific and adds the given string to the message header, according to the relevant protocol specifications for this ASE. It is a specialization of the general $p \rightarrow p'$ transformation in LT2.
- `string pop(message)`
  The pop primitive is also ASE-specific and removes and returns the appropriate portion of the header prefix from the message. It is also a specialization of the $p \rightarrow p'$ transformation in LT2.
- `{<ase, string>} lookup(ase, string)`
  The lookup primitive is used to query the local switching table. It returns the set of exact mappings found.

The specification of the "leads-to" relation in Section 3.2 assumes that the switching table can only be queried through an exact lookup operation to keep the abstract specification simple. In reality, the switching tables can be more efficiently implemented with better data structures. In particular, switching tables may use an aggregated information representation and longest-prefix lookups or provide a *default mapping* which is used for all those names that do not have an explicit mapping in their switching state. These considerations are out of the scope of our work.

4. CONTROL MECHANISMS

With the model introduced so far, we can only describe basic packet forwarding in a static network. However, as mentioned before, each ASE's local switching table needs to be populated with mappings in order to perform forwarding or distributed resolution at any meaningful scale. In addition, network dynamics necessitate dynamic updates to switching tables. We first discuss routing as the basic mechanism to allow forwarding within a single name space. We then study how name spaces can be combined and use this to model on-demand path setup mechanisms.

4.1 Combining Name Spaces

The heterogeneous nature of network environments results in different network technologies with different assumptions, goals, and design strategies. This naturally results in a diversity of approaches for naming network elements. Different naming schemes are appropriate for different requirements. Due to the combination of network technologies, a message may travel through many name spaces before reaching the destination.

Routing is the control process that creates and maintains consistent forwarding *within* a name space in the presence of network dynamics. More formally, the goal of routing is to establish, at some set of ASEs γ, appropriate local switching table entries so that each member of γ has a name for each other member of γ with scope γ. Routing is thus defined based on a name scope, which through the definition of name depends on the "leads-to" relation. Thus, a change in network topology results in a breakage of the "leads-to" relation at some ASEs, which invalidates names. Dynamic routing re-establishes the "leads-to" properties throughout γ. By establishing consistent forwarding for a set of ASEs and names, routing effectively transforms all names into addresses and enables *datagram forwarding*. Invariants of the routing process have been studied elsewhere, for example in [16].

We now consider how to forward messages across name spaces, using either the *overlay* or the *gateway* model.

Overlay In the overlay model, islands of an overlay name space are connected with each other via tunnels across an underlay name space. This corresponds to pushing underlay names on the header stack that enable the transmission of a message from the ingress tunnel endpoint to the egress. The underlay network is oblivious to the overlay naming scheme and thus provides a transparent service between tunnel endpoint. This model uses the encapsulation and decapsulation mechanisms presented in Section 3.5. Examples are TCP over IP, I3 over IP, and MPLS over Ethernet.

Gateway In the gateway model, name spaces are connected by replacing the name from the sender name space with a name that is valid in the receiver name space. This model uses the label switching mechanism from Section 3.5. It requires a path setup mechanism to set up name translation in each sending direction. Examples are NAT, MPLS, or DNS/IP translation.

In both models, names need to be mapped from one name space to another. These mappings can be statically configured, or use on-demand resolution (cf. Section 3.4), such as ARP or DNS, or *path setup* (see below). In contrast to previous work, the "leads-to" relation provides a single framework for both the Overlay and the Gateway model and thus allows analysis of end-to-end communication paths and reachability in compound name spaces.

A communication message that traverses multiple name spaces connected through the overlay model carries multiple source and destination names that characterize the past and future path of the message. It is desirable for the logical stack of destination addresses to be sufficient to reach the destination and at the same time, for the logical stack of source address to be sufficient to reach the source object. We can define this formally:

Deliverability A message $pm@^xA$ is deliverable to B, if and only if $pm@^xA \rightarrow m@^yB$.

Returnability A message $pm@^xA$ is returnable to C, if and only if $\exists r_A(p) : r_A(p)m^xA \rightarrow m@^yC$ where $r_A(p)$ is some suitable transformation function of p at A.

Returnability has been studied previously by Zave [25].

If an overlay peer's name has only local scope and is not sufficient for returnability to that peer, we call it a *weak* name. In this case, it needs to be combined with an underlay ASE's name to form a complete name. For example, a transport layer port (e.g. in UDP or TCP) is only used for local demultiplexing and needs to be combined with an IP address to completely identify the trans-

port instance.[2] In case of a weak overlay name, the delayed pop mechanism introduced in Section 3.5 needs to be used to maintain returnability.

Note that returnability does not require path symmetry. Even if both the overlay and underlay name are complete, that does not guarantee that the return path towards the overlay peer actually goes through the previous peer in the underlay. For example, both IP and MAC addresses are complete, but it is not necessary that the return path to an IP sender uses the same previous MAC hop as the forwarding path.

Also note that local mapping state at intermediate ASEs allows a separation of concerns, in effect, late binding of destination addresses to paths. Compared to a fully-qualified "path name", instead of only naming the successor, each name is then a reference to a path segment. This trade-off has been previously described using the terms 'header state' and 'table state' [4].

4.2 Path Setup

To allow path setup in a compound name space, a protocol header can carry a *path label*, in addition to source and destination names. This label, which is carried by an implicit or explicit connection setup message, allows an alternative specification of returnability (in addition to source addresses) and is used for path setup mechanisms needed for the gateway model of combining name spaces. An example is the MPLS path setup. The path label carried in a call setup message must allow deliverability of its response. In practice, the role of the path label is assumed either by a special field in a protocol header, for example in network control messages, or an existing source identifier is re-used. The operations *swap* and *nop* can be used to modify the path label and seem sufficient to model most existing path setup mechanisms.

Swap - Virtual Circuit In a virtual-circuit network, such as MPLS, the path label is swapped at each forwarding object during call setup. Each ASE provides the successor ASE with a label that is locally unique and creates corresponding switching state, such that a response message from the successor ASE carrying the label is forwarded towards the original initiator. A path setup may involve control messages or may be implicit, for example when a NAT node processes a datagram from the internal to the external network.

Nop - Bridging Self-learning Ethernet bridging and reverse path forwarding are prominent examples of algorithms that do not modify a path label during path setup. The source address of an incoming frame is used as returnability information to update local switching state. Thus, each transmitted frame implicitly triggers a control operation without any modification to the header stack.

It is important to note that a path setup mechanism relies on the availability of naming and forwarding *without* a pre-existing path. In practical terms, this means that path setup requires an existing datagram forwarding mechanism and routing process, which defaults to direct communication.

The path setup mechanisms described above follow the unidirectional receiver-initiated model for path setup where the predecessor ASE assigns a label that the successor ASE can then use for upstream communication. This is the simplest case for path setup, because it requires the fewest assumptions between peers. The alternative, traditional sender-oriented path setup relies on the additional assumption that both predecessor and successor share a name scope, which in turn requires point-to-point communication between peers.

[2]Strictly speaking, a protocol number is also a weak name, but for convenience, many Internet protocol numbers are globally standardized.

4.3 Control Primitives

Similar to the forwarding primitives in Section 3.6, we introduce control primitives that implement the functionality discussed in the previous section. We augment the ASE definition, such that it can update its local switching table and/or create and send a reply message carrying information from the local switching table.

- `update(ase, string, ase, string)`
  The update operation is used to modify the mappings in the local switching table. The first two arguments specify the index and last two arguments the value. Since mappings contain a set, more than one operation would typically be used to manage it. However, we only specify a single primitive that either adds an element to the set or clears the set, if the value arguments are both empty.
- `string getlabel(message)`
  This operation reads and returns the path label, if applicable to the ASE.
- `setlabel(message, string)`
  This operation sets the path label, if applicable.
- `message create(opcode)`
  This primitive creates a new message. A control message is tagged with the appropriate opcode.
- `message response(message, opcode)`
  This ASE-specific operation creates a response message using the available response information (source names and/or path label). In particular, the response primitive chooses the appropriate returnability information from either source names or the path label.

5. PROCESSING

It turns out that the fundamental elements for message processing can be expressed as a small number of processing *patterns*, using the primitives described in Sections 3.6 and 4.3. Those ASEs that exchange explicit control messages tag such messages with abstract *opcodes* that can be thought of as extra names in the protocol header. We use patterns, primitives, and opcodes to abstractly describe a basic ASE in pseudo-code. Actual ASEs are then created by refining the basic ASE, for example through inheritance. This abstract design forms the basis of our implementation architecture. In the prototype, ASEs are implemented as Click elements [14].

5.1 Abstract Processing

The pseudo-code in Figure 2 shows the main processing routine for an ASE. It forwards messages according to the "leads-to" relation, but also invokes control operations when necessary. The `ctl()` primitive returns the control opcode, if the message is a control message. In reality, not all ASEs will require all functionality, so the routine below is that of a conceptual "Super-ASE", from which ASEs can be derived by specialization.

We logically partition the overall processing into several processing patterns and discuss each one below. It is worth noting that this small code snippet is sufficient to model circuit switching, virtual circuit switching, and packet forwarding, as well as path setup and name resolution. This hints at the power of our axiomatic framework. In a real implementation, each pattern would likely be implemented as a separate subroutine.

```
1   process(ase prev, message msg) {
2     bool setup = (ctl(msg) == SETUP
3               || prev in this->SETUP_ASE);
4     string lin, lout;
5     if (setup) lin = lout = getlabel(msg);
6     string n = pop(msg);
7     {<ase, string>} S = lookup(prev, n);
8     if (!S && this->RESOLVE_ASE) {
9       resolve(n); // wait for S update
10      S = lookup(prev, n);
11    }
12    for each <ase, string> s_i in S {
13      if (s_i.ase == this) { // local
14        if (ctl(msg) == RLOOKUP) {
15          respond(prev, msg, n, s_i.string);
16        } else if (ctl(msg) == RUPDATE) {
17          rupdate(msg);
18        } else {
19          // other local control activity
20        }
21      } else { // forward
22        message outmsg = copy(msg);
23        push(outmsg, s_i.string);
24        if (setup) {
25          if (VC) lin = local_name(prev, n);
26          update(s_i.ase, lin, prev, lout);
27          setlabel(outmsg, lin);
28        }
29        send(s_i.ase, outmsg);
30      }
31    }
32  }
```

Figure 2: Main Processing Routine

5.2 Processing Patterns

The ***Forward*** pattern is used for regular forwarding of data messages and can be found in lines 6,7,12,22,23,29 of `process()`. In Section 6.2, we give a proof in a formal logic that this code implements the forwarding functionality described in Section 3.5.

The ***Setup*** pattern is used for forwarding path setup requests. An ASE may be configured, so that ***Setup*** is the default action when messages arrive from certain predecessor ASEs, which are collectively termed SETUP_ASE for that ASE. Alternatively, ***Setup*** is triggered by a special control message carrying the SETUP opcode. This is shown in `process()`, lines 2-5. In lines 24-28, the boolean `VC` value is another configuration parameter and determines whether the setup operation will implement the Virtual Circuit or Bridging version of path setup, as introduced in Section 4.2. In the VC case, `local_name(prev, n)` either determines which local name is currently used to replace `<prev, n>`, or finds and assigns a new available local name. This can be done with a separate table, or by indexing the value side of the local switching table.

The ***Resolve*** pattern initiates a remote resolution request, if necessary, and swaps or pushes the resulting name into the message. We mandate that a resolving ASE has a predefined successor ASE, termed RESOLVE_ASE, to which it forwards resolution requests. In the pseudo-code in lines 8-11, if a name cannot be resolved locally, distributed resolution is invoked. This is done using the `resolve()` function, shown below, which creates an RLOOKUP message, using the requested name as destination address.

```
1   resolve(string n) {
2     message outmsg = create(RLOOKUP);
3     push(outmsg, n);
4     send(this->RESOLVE_ASE, outmsg);
5     wait_for_rupdate(n);
6   }
```

We eliminate the asynchrony typically involved with distributed resolution, by modelling `resolve()` as a synchronous procedure that blocks until a name is resolved. This is expressed by line 5, which waits on a condition for the particular name. The corresponding signal is sent by `rupdate` (see below).

The ***Response*** pattern is used when a remote lookup request arrives, detected through the RLOOKUP opcode, and the requested name is found in the local switching table.[3] The branch is shown in `process()` in line 14 and 15. The actual response functionality is shown in the helper function `respond()` below. The `add_data()` primitive models the reply part of the name resolution protocol, the details of which are currently outside the scope of our framework.

```
1   respond(ase next, message m, string n1, n2) {
2     message outmsg = response(m, RUPDATE);
3     add_data(outmsg, n1, n2);
4     send(next, outmsg);
5   }
```

The ***RUpdate*** pattern is invoked on receipt of a response to a resolution request. The corresponding branch is shown in lines 16 and 17 of `process()`. The code for `rupdate()` is shown below.

```
1   rupdate(message msg) {
2     string n1 = get_data(msg);
3     string n2 = get_data(msg);
4     if (!this->RESOLVE_SWAP) n2 = n1 + n2;
5     update(*, n1, this->FORWARD_ASE, n2);
6     signal_to_resolve(n1, n2);
7   }
```

After a resolution reply has been received, its information is extracted using the `get_data()` primitive (lines 2-3), which corresponds to the `add_data()` primitive introduced before. Depending on the resolver type, the configuration parameter RESOLVE_SWAP determines whether a name is replaced after translation or whether the resolved name is prepended during forwarding (line 4) and local switching state is updated accordingly (lines 5-6). The previous ASE field is left unspecified and matches any later request. Finally, the condition for this name is signalled (line 7), such that `resolve` can resume its operation.

6. FORMAL SEMANTICS

We present the formal semantics of the primitive operations of Sections 3.6 and 4.3 in an axiomatic style using logical assertions [10]. This proof framework is paradigmatic in providing a rigorous basis for verification and formal proofs of correctness of protocols and their implementations, and justifies calling our framework "axiomatic". This style allows us to specify and verify the behaviour of networks *abstractly* by using properties specified in a logic, rather than in terms of lower-level *operational* descriptions using *abstract machines*. Being a symbolic proof system that supports compositional reasoning, it is well suited for machine-assisted proofs. While assertional style and temporal logic techniques have been used in reasoning about protocol designs and their composition [7,

[3]We store both switching and translation mappings in the switching table.

25], we are unaware of any previous work that has presented Hoare-style axiomatic rules for reasoning about network connectivity, and believe this to be a novel contribution.

Underlying mathematical domains. For reasoning about the behaviour of ASEs, we assume that we have a sound and complete axiomatization of the underlying multi-sorted algebra, comprising:

- a type *string* with concatenation operator.
- messages, of type *message*, with special operations *setlabel* and *getlabel* for setting and extracting labels in messages. These satisfy the equation
  getlabel(*setlabel*(m,n)) = n for all *message* m, *string* n.
- pairs of type $ase \times string$, with pair-forming and projection operations $\langle \_, \_ \rangle$, $\_.ase$ and $\_.string$ respectively, with the standard equational properties.
- the type of *Control Opcodes*.
- switching tables with entries drawn from the above data types, and with the operation $T[u \mapsto v]$ of augmentation of T with the entry $v \in T[u]$.

Assertion Language. Properties or assertions concerning elements drawn from these data types are expressed in a subset of a first-order language with equality over expressions, including those with variables, denoting objects from these data types. Let metavariable θ (possibly with subscripts and superscripts) range over first-order formulas in this language. We assume familiarity with first-order logic, particularly with the notion of substituting an expression e for all *free* occurrences (i.e., those not bound by a quantifier) of a variable x in a formula θ, which we write as $\theta[e/x]$.

In addition to this first-order language, we assume the definition of the "leads-to" relation given in Section 3.2. Behavioural properties concerning connectivity are expressed in a limited extension of the first-order language with this inductively defined predicate: Let φ be a typical formula in this language:

$$\varphi ::= \theta \mid m@y \rightarrow m'@z \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \theta \supset \varphi \mid \forall x.\varphi_1 \mid \exists x.\varphi_1$$

Substitution is extended to formulas of the form φ.

Note that we allow sub-formulas of the form $m@y \rightarrow m'@z$ only in *positive* positions, i.e., those not under any explicit or implicit negation. This is because the "leads-to" relation is inductively defined, and therefore requires *monotonicity* in its interpretation. Note that all the concepts we have defined so far (deliverability, name for an ASE, address, scope, etc.) satisfy this condition.

6.1 A Hoare-style Logic with "Leads-To"

The axioms and inference rules are given in terms of *Hoare triples* $\varphi_1 \ \{P\} \ \varphi_2$, where P is a statement or program fragment, and φ_1 and φ_2 are called a *pre-condition* and *post-condition* respectively. The interpretation is that if precondition φ_1 holds, and P is executed to completion, then post-condition φ_2 holds in the resulting state. Note that in the case of communication networks, the notion of "state" has two component classes: (i) The internal state of local variables within each ASE, which affects and is affected by the local program execution; this is abstractly captured by formula of the form θ. (ii) The network connectivity state, which is distributed by nature and affected by the contents of the switching tables and by the communication primitives `send` and `receive`. The program P is written using *statements* in the language comprising the primitives, enhanced with assignment, sequencing, a "for all" construct and conditional constructs.

The semantic framework is presented using inference rules of the form $\frac{\mathscr{A}}{\mathscr{B}}$, where $\mathscr{B}$ is a single Hoare triple, called the *conclusion*, and $\mathscr{A}$ consists of 0 or more Hoare triples, called *assumptions*. The interpretation of the rules is that if *all* assumptions hold, then the conclusion also holds. Rules with no assumptions are called *axioms*.

Assignment	$\dfrac{}{\varphi[e/x] \ \{\mathtt{x = e}\} \ \varphi}$
Sequential	$\dfrac{\varphi_1 \ \{P_1;\} \ \varphi_2 \quad \varphi_2 \ \{P_2;\} \ \varphi_3}{\varphi_1 \ \{P_1;P_2;\} \ \varphi_3}$
IfThenElse	$\dfrac{(\varsigma \wedge b) \ \{P_1\} \ \varphi \quad (\varsigma \wedge \neg b) \ \{P_2\} \ \varphi}{\varsigma \ \{\mathtt{if}\ b\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2\} \ \varphi}$
IfThen	$\dfrac{(\varsigma \wedge b) \ \{P_1\} \ \varphi \quad (\varsigma \wedge \neg b) \supset \varphi}{\varsigma \ \{\mathtt{if}\ b\ \mathtt{then}\ P_1\} \ \varphi}$
ForEach	$\dfrac{\varsigma[c_i/c] \ \{P\} \ \varphi_i \quad \text{for each } c_i \in S}{\varsigma \ \{\mathtt{for\ each}\ c\ \mathtt{in}\ S \ \{P\}\} \ \bigwedge_i \varphi_i}$
Consequence	$\dfrac{\varphi_1 \ \{P\} \ \varphi_2 \quad \vdash \varphi_1' \supset \varphi_1 \quad \vdash \varphi_2 \supset \varphi_2'}{\varphi_1' \ \{P\} \ \varphi_2'}$

Figure 3: Hoare Logic Rules for a Simple Sequential Language

Figure 3 lists the rules that we inherit or adapt from Hoare logic for traditional sequential imperative languages [10]. Observe that the "Consequence" rule allows us to strengthen preconditions and weaken post-conditions, and is vital in that it connects the mechanistic syntax-directed approach of manipulating formulas to the mathematics of the underlying domain, and supports modularity and compositionality of the framework.

We focus on the rules particular to our primitives, which we state as axioms in Hoare logic. The next set of rules formalize the primitives that operate locally on messages.

- $n \ = \ \mathtt{pop}(m);$

  $$\overline{m = pm' \wedge \varphi[m'/m, p/n] \ \{n \ = \ \mathtt{pop}(m);\} \ \varphi}$$

- $w \ = \ \mathtt{copy}(m);$ $\quad \overline{\varphi[m/w] \ \{w \ = \ \mathtt{copy}(m);\} \ \varphi}$

- $\mathtt{push}(m,n)$ $\quad \overline{\varphi[nm/m] \ \{\mathtt{push}(m,n)\} \ \varphi}$

The operations `pop` and `copy` have been embedded in an assignment statement. The rule for the mutable operation `pop` employs concatenation of p to m' to decompose message m.

Primitives `lookup` and `update` operate locally on the switching table at an ASE, assumed to be A in the following rules.

- $S \ = \ \mathtt{lookup}(B,n);$

  $$\overline{\varphi[S_A[B,n]/S] \ \{S \ = \ \mathtt{lookup}(B,n);\} \ \varphi}$$

  The set of table entries in $S_A[B,n]$ replace the set-valued variable S in the property to be shown.

- $\mathtt{update}(B,n,C,p)$

  $$\overline{\varphi[S_A[\langle B,n\rangle \mapsto \langle C,p\rangle] \ \{\mathtt{update}(B,n,C,p)\} \ \varphi}$$

  If assertion φ is to hold after the update of S_A, then in the prior state, it should hold of a table that was equal to $S_A[\langle B,n\rangle \mapsto \langle C,p\rangle]$.

The communication primitives `send` and `receive` operate on the "leads-to" relation. The following rules are assumed to operate at ASE A.

- $(X,m) = \texttt{receive}();$

$$\overline{\varphi[m'/m,B/X]\ \{(X,m) = \texttt{receive}();\}\ \varphi \wedge \varsigma}$$

where $\varsigma = m'@B^A \rightarrow m@^X A$. This rule implements LT1, and otherwise resembles an assignment of values to variables.

- $\texttt{send}(C,m)$ $\quad \overline{\varphi_1\ \{\texttt{send}(C,m)\}\ \varphi}$

where $\varphi = \theta \supset m'@x \rightarrow m''@y$ and $\varphi_1 = \theta \supset (m'@x \rightarrow m''@y \vee (m'@x \rightarrow m@A^C \wedge m@A^C \rightarrow m''@y))$. For convenience, we assume that φ is of the form $\theta \supset m'@x \rightarrow m''@y$. This is not a serious limitation, since the formulas in which we are interested can be converted via De Morgan laws and other meaning-preserving logical transformations in a logical combination of formulas of this form or of form θ, which can then be individually handled.

We "split" any $m@x \rightarrow m'@y$ formula into two disjuncts, one of which is merely a copy whereas in the other we interpose $m@A^C$. The idea is a symbolic realization of the fact that either the original formula held irrespective of this send operation, or it is achieved because of this send operation. This intuition is based on the algebra of *paths* — there is a path from x to y either if there is a direct path not going through A, or else there is a path from x to A, and a path from A to y.

- The rules for $\texttt{getlabel}(m)$ and $\texttt{setlabel}(m,n)$ are quite standard.

$$\overline{\varphi[getlabel(m)/l]\ \{l = \texttt{getlabel}(m);\}\ \varphi}$$

$$\overline{\varphi[setlabel(m,n)/m]\ \{\texttt{setlabel}(m,n)\}\ \varphi}$$

For lack of space, we omit the rules for the remaining primitives such as `ctl`, `create` and `response`. These are standard and do not differ greatly in structure from the rules for the message operations.

6.2 An Example Proof: Correctness of *Forward*

We illustrate the power of the formalization by presenting an annotated proof of correctness of the ***Forward*** pattern of Section 5.2. The code in Figure 4 is extracted from the pseudo-code of Section 5 by considering the program partition (or conditioned slice) dealing with pure forwarding, and removing typing information. Program *slicing* and partitioning [23] are increasingly gaining currency as pragmatic techniques used in the verification of large software.

The desired post-condition is that if a message pm_0 is at the ASE A's input port from B, then a transformed message is place at each of A's corresponding outgoing port for each entry for $\langle B,p\rangle$ in A's switching table, formally written as:

$$\bigwedge_{s_i \in S} \supset (pm_0@^B A \rightarrow (s_i.string)m_0@A^{s_i.ase})$$

By predicate transformation of the post-condition through the program in the backward direction, using the axioms given above, we get a *verification condition* that

$$s \in S_A[prev,p] \supset (pm_0@^B A \rightarrow (s.string)m_0@A^{s.ase} \wedge (s.string)m_0@A^{s.ase} \rightarrow (s.string)m_0@A^{s.ase})$$

follows from the preconditions $prev = B \wedge m = pm_0 \wedge \exists^B A$ This is easy to establish from the definition of "leads-to" and by LT2 and

$--$ Pre : $prev = B \wedge m = pm_0 \wedge \exists^B A$

$s \in S_A[prev,p] \supset (pm_0@^B A \rightarrow (s.string)m_0@A^{s.ase} \wedge (s.string)m_0@A^{s.ase} \rightarrow (s.string)m_0@A^{s.ase})$

(**6**) $\texttt{n} = \texttt{pop}(\texttt{m});$

$s \in S_A[prev,n] \supset (pm_0@^B A \rightarrow (s.string)m@A^{s.ase} \wedge (s.string)m@A^{s.ase} \rightarrow (s.string)m_0@A^{s.ase})$

(**7**) $\mathbf{S} = \texttt{lookup}(\texttt{prev},\texttt{n});$

$s \in S \supset (pm_0@^B A \rightarrow (s.string)m@A^{s.ase} \wedge (s.string)m@A^{s.ase} \rightarrow (s.string)m_0@A^{s.ase})$

(**12**) `for each s in` $\mathbf{S}$ {

$s_i \in S \supset (pm_0@^B A \rightarrow (s_i.string)m@A^{s_i.ase} \wedge (s_i.string)m@A^{s_i.ase} \rightarrow (s_i.string)m_0@A^{s_i.ase})$

(**22**) $\texttt{outmsg} = \texttt{copy}(\texttt{m});$

$s_i \in S \supset (pm_0@^B A \rightarrow (s_i.string)outmsg@A^{s_i.ase} \wedge (s_i.string)outmsg@A^{s_i.ase} \rightarrow (s_i.string)m_0@A^{s_i.ase})$

(**23**) $\texttt{push}(\texttt{outmsg},\texttt{s.string});$

$s_i \in S \supset ((pm_0@^B A \rightarrow outmsg@A^{s_i.ase} \wedge outmsg@A^{s_i.ase} \rightarrow (s_i.string)m_0@A^{s_i.ase}) \vee \ldots)$

(**29**) $\texttt{send}(\texttt{s.ase},\texttt{outmsg});$

$s_i \in S \supset (pm_0@^B A \rightarrow (s_i.string)m_0@A^{s_i.ase})$

(**31**) }

$--$ Post : $\bigwedge_{s_i \in S} \supset (pm_0@^B A \rightarrow (s_i.string)m_0@A^{s_i.ase})$

Figure 4: Correctness Proof of *Forward*

LT4. For readability, we have omitted the irrelevant disjunct that arises from `send`.

6.3 Towards Verified Protocols

Similarly, we can verify the correctness of the ***Setup*** pattern, establish in terms of a *returnability* property. This involves showing that if a message $pm@^x A$ is returnable to B, and if $pm@^x A$ leads to a message $p'm@^y C$ that is returnable to A, then (provided path setup is requested), by the action of the ***Setup*** pattern on A's switching table, the message $p'm@^y C$ is returnable to B. The proof relies on the existence of transformation $r_A(\_)$ which is *constructively* demonstrated by the action of ***Setup***.

From the correctness of ***Forward*** and ***Setup***, it becomes possible to show the implementations of *overlay* and *gateway* compositions of name spaces are correct. In the overlay case, the proof involves showing that tunnels between ASEs in the scope of the underlay name space induce tunnels between certain ASEs in the scope of the overlay name space. A crucial lemma in verifying traditional layered architectures is that the overlay and underlay name spaces are disjoint. Additional facts used in the proofs are that processing at each ASE divides a name space into *finitely many* classes (based on its neighbourhood connectivity); furthermore, we exploit the uniformity in how names in the same class are mapped in the tables.

The correctness of the name space compositions provide modularity results that can assist in the proofs of correctness of, for instance, the NAT example of Section 7.1. Here we need to establish the correctness of e.g., the TCP-IP and IP-Ethernet overlays, the NAT gateway composition, and that, e.g., adjacent Ethernet objects share a name space.

We intend to verify a large variety of communication protocols using our techniques. Future work on the theoretical foundations includes a formal proof of soundness of the axiomatic semantics

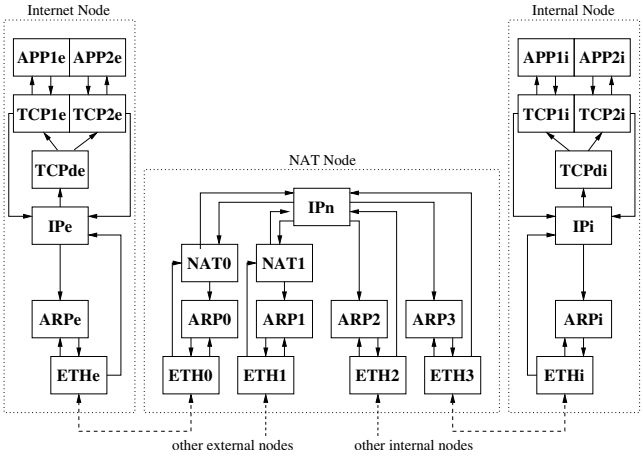

Figure 5: NAT Example - Overview

Table 1: NAT Example - Lookup Table (θ = proto num)

ASE	Index Name p	Value Name p'
TCP	IPdst/src, TCPdst/src	IPdst/src, TCPdst/src
IP	IPdst	IPgw, IPdst/θ
up	IPdst/θ	IPdst
NAT	IPdst/src/θ, TCPdst/src	IPdst/src'/θ, TCPdst/src'
up	IPdst/src/θ, TCPdst/src	IPdst'/src/θ, TCPdst'/src
ARP	IPgw	MACdst/θ
up	ctl opcode	N/A (ctl response)
ETH	MACdst	MACdst/src
up	MACdst/θ	$\varnothing$

with respect to operational semantics. While we have informally checked the soundness, such a result is best substantiated with the help of a formal machine-checked proof using a theorem prover. We also intend to explore the expressive power of the logic, and investigate its relative completeness.

7. EXAMPLES

7.1 TCP over NAT

We outline a non-trivial example of TCP over IP over Ethernet communication in the presence of a NAT network element along the IP path. The setup of ASEs and direct communication is shown in Figure 5. TCPdi and TCPde refer to TCP demultiplexers, and TCPi and TCPe are the actual TCP instances. Table 1 gives an overview of the header fields that are used to index the local switching tables and the values that are returned. If applicable, the first row for an ASE shows the used fields for messages arrived through connections leading to the ASE in downward direction and the second one for those in upward direction, as shown in Figure 5. The ASE entries of the switching table are omitted for brevity.

Application ASEs are modelled as black boxes that create and consume messages. Because TCP ports are weak names and are valid only in the context of IP addresses, a sending TCP ASE must include the IP addresses into the message header and the receiving TCP demultiplexer also uses the IP addresses to demultiplex to individual TCP objects. For outgoing packets, IP prepends the gateway's IP address to the packet, which is later resolved by ARP into the MAC destination address. ARP instances communicate with each other using RLOOKUP and RUPDATE to perform remote ARP lookups. In addition to forwarding outgoing packets, a NAT module also performs a path setup following the ***Setup*** pattern from Section 5. The whole five-tuple of IP and transport names is used as a path label, although only the source port is used to find a free local name. Because outgoing packets need to be assigned the proper IP source address, IP forwarding must happen before NAT processing and it is easiest to model one NAT instance per external interface. On the return path, packets need to be processed by NAT first, since they must have the internal destination address restored, before being forwarded by IP. In Table 2, we summarize the other necessary configuration values for this example.

Table 2: NAT Example - ASE Configuration

ASE	Config Parameter	Value
NAT class	VC	true
NAT0/1 objects	SETUP_ASE	IPn
ARP class	RESOLVE_SWAP	false
ARPx objects	RESOLVE_ASE	ETHx

7.2 Other Examples

We now sketch key some other scenarios that can be described using our framework.

- **DNS** DNS conceptually permits two operations: (a) the installation of a translation and (b) the use of this translation for resolving DNS names to IP addresses. The manual registration of a DNS name is outside the scope of our work. Once installed, this translation is reached by *DNS-name-based* forwarding of a resolve request to the appropriate ASE, which elicits a response with the translation, causing the installation of local state using the ***RUpdate*** pattern.

- **Hierarchical Mobile IP** The key concern of any host mobility protocol is the registration of the mobile node along a hierarchy of anchor points and subsequent packet forwarding along anchors to the mobile host. This can modelled as the bridging path setup pattern between mobility agents and anchor points.

- **I3** Using our framework, I3 [21] becomes a straightforward implementation of "leads-to" with Chord [22] as the routing process. When the outermost I3 identifier is exactly matched in the local switching table, the message is potentially replicated. For each copy, the outermost identifier is replaced by a stack of identifiers (which may be empty) and fed back into the I3 ASE, or, if all I3 identifiers are removed then, the message is forwarded to the IP ASE. If there is no match, the message is forwarded according to Chord forwarding rules.

8. PROTOTYPE

Besides formally studying the axioms using Hoare logic, we have implemented a prototype *Universal Forwarding Engine (UFE)* in the Click environment [14], a framework for building flexible, configurable routers. A Click router configuration is specified by a directed graph of *Click elements* where each element defines a simple router function such as queueing, scheduling, or updating a field of a packet. Each ASE is implemented as a Click element. However, unlike most traditional Click elements, an ASE represents a more complex unit of processing, such as IP forwarding, rather than just a simple function. Click elements are implemented in C++ and form the core of the Click runtime system. The system is configured at runtime by interpreting a configuration file, which specifies how and which elements are connected to each other.

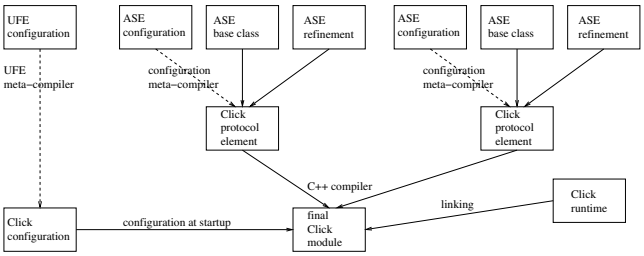

Figure 6: UFE Development in Click

We use a hybrid approach of inheritance and meta-compilation to produce the Click implementation and configuration. The complete development process is shown in Figure 6. The core element of the UFE is an *ASE base class*, which provides implementations for all processing patterns. For rapid prototyping, the ASE base class provides templates for common data structures and all operational primitives. The base class also provides helper functions that are needed for practical protocol implementation. When combined with an ASE configuration file, our base class is sufficient to generate a C++ implementation of an ASE as a Click element. In addition, parts of the ASE base class can be refined using inheritance, by providing protocol-specific implementations of data structures and primitives, or even patterns. This allows for a gradual transformation of an initial functional protocol prototype, which is simple and analyzable, into a more realistic and efficient implementation.

We summarize below the information and operations that need to be adapted to each ASE:

- **Names and State** It is necessary to specify a *name mask* and a *label mask* to describe which header fields are to be transformed into the forwarding prefix and, if applicable, which are used as the path setup label. Also, the *response* transformation must be documented for those ASEs that perform distributed resolution. Without further specialization, the generic switching table contains just mappings from a pair of ASE-and-string to a pair of ASE-and-string. For efficiency, one can refine the table format, along with the *lookup* and *update* operations, and use specialized data structures.

- **ASE Type Configuration** `VC` and `RESOLVE_SWAP` specify how an ASE type implements the ***Setup*** or ***Resolve*** pattern, respectively.

- **ASE Object Configuration** The variable `SETUP_ASE` contains a list of ASEs for which ***Setup*** is the default action and `RESOLVE_ASE` defines the successor ASE to which remote resolution requests request are sent.

- **Protocol Glue** A protocol element needs to be configured and/or refined to adhere to standardized protocol specifications and mechanisms. At the very least, the system needs to know how to add or remove a protocol header and what the header looks like. In addition, we provide hooks for pre- and post-forwarding operations to incorporate those protocol mechanisms that we cannot yet express with our abstract framework.

Our prototype system compiles meta-language programs composed from the primitives in Sections 3.6 and 4.3 to Click elements in C++. We also compile from a configuration file to a corresponding Click configuration file. Our meta-compilers were developed using lex and yacc: the configuration meta-compiler is 150 lines of yacc code, and the element meta-compiler is 158 lines of yacc code. Our Ethernet bridge, described in 21 lines of meta-language compiled to 91 lines of C++ in a Click element. When compiled into the Click runtime, we were able to successfully bridge Ethernet frames between two segments. We have also built, using the meta-language, an IP forwarder, and a NAT middle-box.

9. DISCUSSION AND CONCLUSION

Our work brings together three research threads that span the networking and formal verification communities. The first thread is that of using formal notation to compactly and precisely model network communication, and, in particular, communication in the Internet. This allows us to derive elementary axiomatically-sound forwarding and control operations. Second, we exploit standard techniques in formal verification to prove the correctness of network protocols composed from these operations. Finally, our work builds on extensive past work in rapid protocol prototyping. We use meta-compilation to translate from a protocol expressed in terms of elementary operations to a C++ implementation that can be embedded in the Click engine and incorporated into the Internet.

It is illuminating to compare our definition of names and addresses to that in common use. Commonly, the name of an object is a 'human-readable' string, and the address is a 'machine-readable' location. Names are meant to refer to entities, and addresses are meant to get to them. This is, of course, incorrect when faced with name-based forwarding and address masquerading. By tightly coupling the act of communication with the concept of a name and precisely specifying the scope of a name, we not only achieve conceptual clarity but also can resolve a host of conceptual pitfalls. For instance, an IP address *is* an address in the public IP scope, but devolves to a name within a NAT gateway that bridges name scopes. Similarly, a "toll-free" number *is* an address until it reaches a translation table, that then translates it to *another* address in the same scope. These distinctions are possible because of the simplicity and clarity of our axiomatic framework.

Our axiomatic approach allows us to discover heretofore hidden isomorphisms. For instance, it is easy to see that NAT is essentially the same as RSVP-TE or ATM, in that it sets up a forwarding table that translates between name spaces. These name spaces are the public and private IP name spaces, which are, in principle, similar to the VCI spaces in ATM. Even more interestingly, the same bridging of name spaces is accomplished in Mobile IP, IP multicast and I3. Therefore, the same protocol engine, with minor modifications, can be used to implement these protocols! From a more pragmatic perspective, implementation optimizations for any of these protocols apply equally to all.

Another interesting consequence of our work is the potential to automatically build validated protocol implementations, perhaps even in hardware. We can do so by expressing a protocol in terms of elementary operations, and then using a Hoare-logic theorem prover to prove its correctness. The same protocol can then be meta-compiled, perhaps to an FPGA. This would eliminate error-prone human coding of complex protocols.

Finally, we have confined our analysis to a system where the only dynamic operations (other than message transfer itself) is updating tables within an ASE. It is possible to extend this in two ways. First, we can consider the installation of a new ASE instance in the network. This would allow the network to update itself in response to observed network behaviour. Second, we could allow new ASE types to be defined and then installed in the network. This would allow our framework to also be a basis for active networking.

Although we recognize the power of our axiomatic framework, we also realize that it has several significant limitations. In gen-

eral, we do not yet include performance or quality in the model, as previously mentioned in Section 1. In short, our approach does not consider time, errors, and physical limits. We address each next.

- First, our system is oblivious to time, other than to have a resolution request block awaiting response. This prevents us from modelling timeouts, retransmissions, etc.
- Second, we assume that all transmissions are error free. So, we cannot model many interesting phenomena driven by packet loss. We speculate that this can be addressed by a *probabilistic "leads-to"* relation as a natural generalization of the "leads-to" relation.
- Finally, we assume that a message that arrives at a port will always find room in a buffer and that the translation table is infinite. Of course, in reality these are limited and the limits can affect protocol correctness.

Other examples of real-world artifacts not covered by the framework are parallel links, normally used for performance or reliability, or the IP TTL mechanism, which is used to mitigate the effect of routing problems.

Due to these limitations, we are unable to express issues relating to network performance, such as packet loss, network congestion, routing oscillations and packet retransmission. We are keenly aware of these limitations and hope to address them in future work.

10. ACKNOWLEDGEMENTS

This work has been supported by the National Science and Engineering Research Council of Canada, the Canada Research Chair Program, and Intel Corp.

11. REFERENCES

[1] B. Ahlgren, M. Brunner, L. Eggert, R. Hancock, and S. Schmid. Invariants: A New Design Methodology for Network Architectures. In *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture, Portland, OR, USA*, pages 65–70, August 2004.

[2] B. Ahlgren, L. Eggert, B. Ohlman, J. Rajahalme, and A. Schieder. Names, Addresses and Identities in Ambient Networks. In *Proceedings of the 1st ACM Workshop on Dynamic Interconnection of Networks, Cologne, Germany*, pages 33–37, September 2005.

[3] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A Layered Naming Architecture for the Internet. In *Proceedings of SIGCOMM '04, Portland, OR, USA*, pages 343–352, August 2004.

[4] G. Chandranmenon and G. Varghese. Trading Packet Headers for Packet Processing. *IEEE/ACM Transactions on Networking*, 4(2):141–152, 1996.

[5] D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *Proceedings of SIGCOMM '88, Stanford, CA, USA*, pages 106–114, August 1988.

[6] K. Fall. A Delay-Tolerant Network Architecture for Challenged Internets. In *Proceedings of SIGCOMM '03, Karlsruhe, Germany*, pages 27–34, August 2003.

[7] Mohamed G. Gouda. *Elements of network protocol design*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[8] T. G. Griffin and J. L. Sobrinho. Metarouting. In *Proceedings of SIGCOMM '05, Philadelphia, PA, USA*, pages 1–12, August 2005.

[9] M. Gritter and D. R. Cheriton. An Architecture for Content Routing Support in the Internet. In *Proceedings of USITS 2001, San Francisco, CA, USA*, pages 37–48, March 2001.

[10] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[11] N.C. Hutchinson and L.L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[12] D.B. Johnson and D.A. Maltz. Dynamic Source Routing in Ad Hoc Wireless Networks. In Tomasz Imielinski and Hank Korth, editors, *Mobile Computing*, pages 153–181. Kluwer Academic Publishers, 1996. Chapter 5.

[13] M. Karsten, S. Keshav, and S. Prasad. An Axiomatic Basis for Communication. In *5th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets V), Irvine, CA, USA*.

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

[15] X. Liu, C. Kreitz, R. Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable. Building Reliable, High-Performance Communication Systems from Components. In *Proceedings of SOSP 1999, Kiawah Island, SC, USA*, pages 80–92, December 1999.

[16] B. Loo, J. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *Proceedings of SIGCOMM '05, Philadelphia, PA, USA*, pages 289–300, August 2005.

[17] C. Partridge, T. Mendez, and W. Milliken. RFC 1546 - Host Anycasting Service, November 1993.

[18] J. H. Saltzer. Naming and Binding of Objects. *Lecture Notes in Computer Science*, 60:99–208, 1978.

[19] M. Schroeder, A. Birrell, M. Burrows, H. Murray, R. Needham, T. Rodeheffer, E. Satterthwaite, and C. Thacker. Autonet: a High-speed, Self-configuring Local Area Network Using Point-to-point Links. *IEEE Journal on Selected Areas in Communications*, 9(8):1318–1335, 1991.

[20] J. Scott, P. Hui, J. Crowcroft, and C. Diot. Haggle: A Networking Architecture Designed Around Mobile Users. In *Proceedings of the 3rd Annual Conference on Wireless On demand Network Systems and Services (WONS 2006), Les Mnuires, France*, January 2006.

[21] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. *IEEE/ACM Transactions on Networking*, 12(2):205–218, April 2004.

[22] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *IEEE Transactions on Networking*, 11:17–32, February 2003.

[23] F. Tip. *A Survey of Program Slicing Techniques*. Centrum voor Wiskunde en Informatica, 1994.

[24] Y. Wang, J. Touch, and J. Silvester. A Unified Model for End Point Resolution and Domain Conversion for Multi-Hop, Multi-Layer Communication. Technical Report ISI-TR-590, USC/ISI, June 2004.

[25] P. Zave. Compositional Binding in Network Domains. In *Proceedings of the 14th International Symposium on Formal Methods, Hamilton, Canada*, pages 332–347, August 2006.