Naming, Addressing, and Forwarding Reconsidered

S. Keshav
School of Computer Science, University of Waterloo
Waterloo, ON, Canada, N2L 3G1

Abstract— **We re-examine the concepts of naming and addressing and show that names are often confused for addresses because of the phenomenon of *masquerading*. Clearly separating the two allows us devise a generalized forwarding algorithm and decompose it into a small set of simple forwarding primitives. We show that forwarding schemes chosen from a variety of systems and at different layers of the OSI hierarchy can be economically expressed using these primitives. We conclude with some ideas on how efficient implementation of these primitives can therefore be used to construct complex, yet efficient, forwarding engines.**

I. Introduction

Naming, addressing, and forwarding are deceptively simple concepts. Classically, a name identifies an entity, usually in a human-understandable fashion [3, 5] and its address locates it. Forwarding is the process by which a switch or router, on receiving data, uses a routing table to decide where to forward the data, then actually forwards it. However, on deeper examination, names and addresses start to look very like each other, and the apparently straightforward process of forwarding turns out to be surprisingly complex.

In this paper, we present a simple framework to understand these concepts and their relationship to each other (Section II-IV). We use this understanding to motivate the notion of generalized forwarding (Section V) that reconciles concepts in traditional IP forwarding with the more exotic forms of forwarding found in peer-to-peer networks and Mobile IP. This leads to the design of a set of forwarding primitives that can succinctly express very different forwarding schemes implemented at different layers of the OSI hierarchy (Section VI-VII). We conclude with a sketch of how efficient implementation of these primitives can be used to construct arbitrarily complex, yet efficient, forwarding engines (Section VIII).

Our contributions are fourfold. First, we provide a simple conceptual framework to distinguish between names and addresses. Second, we construct a generalized forwarding algorithm that motivates the choice of a small set of forwarding primitives. Third, we show how these primitives can be used to economically describe a variety of forwarding schemes in the literature. Finally, we identify a set of hardware accelerators that can be used to improve the performance of a range of forwarding algorithms.

II. Naming, addressing, and forwarding in traditional networks

A. The postal network: hierarchy and default routing

The postal network is one of the earliest communication networks and it is illustrative to study naming, addressing a forwarding in its context.

This research was supported by grants from the National Science and Engineering Council of Canada, the Canada Research Chair Program, Nortel Networks, Sun Microsystems Canada, Intel Corporation, and Sprint Corporation.

The postal system has no names, only addresses. These correspond to physical locations such as mailboxes and post-office boxes. Postal addresses are typically assigned hierarchically, so that the address of a node typically left-extends the address of its parent node (this rule breaks down at the lowest level of the hierarchy). When followed, this rule allows us to represent all the children reachable from any node succinctly. For example, all postal destinations in Waterloo, Ontario, Canada left-extend the string ”Waterloo, Ontario, Canada”, so they can be referred to concisely as the aggregate ”*,Waterloo, Ontario, Canada”. Because of this compact representation, given an item to deliver, a post office can easily determine whether it should be forwarded to a child or peer node–if a suffix of the destination address has a longer match with the address of the child or peer than itself–or to a parent–otherwise (this is the ’default route’). At the lowest level, a post office has to maintain and look up an explicit database of all locally reachable endpoints because the address aggregation rule does not apply.

We make a few observations about the postal system:
1. The only forwarding information needed at an internal node are the addresses of its peers and children. This information is hand-configured without the need for a routing protocol.
2. The ‘default route’ to a parent on an unmatched address allows a post office in Canada to receive a letter addressed, say, in Japanese, and pass it up the hierarchy to the national post office, where the only information needed for correct delivery is that Japanese-addressed letters should be sent to Japan. In other words, forwarding is correctly accomplished despite the fact that only the main post office in Canada understands Japanese. This illustrates the power of default routing.

B. The telephone network: masquerading

The telephone network, in the earliest days, followed the postal model. A call’s destination phone number (its E.164 address) was matched with the local ’exchange number’ or ’area code’ of a phone exchange to route calls. Indeed, in the early days, every central exchange in every area code had a one-hop path to every other area’s central exchange, precisely so that the forwarding decision involved minimal electromechanical switching logic. Note also that in the early phone network, the telephone number, or address, was a physical location corresponding to an actual phone instrument.

With the advent of computer-controlled telephone switches, forwarding logic could be made more complex. A toll-free number, such as 1 800 555 1212 looks like a phone number. However, when it is dialed, the local exchange forwards it to the ’800 area’ phone exchange, which then looks up a table to determine where the call is actually supposed to go to. In other words, the 800 area is really a way to request an additional level of indirection. This indirection table can be changed with the time of

day, day of year, or in more complex ways, for instance, to forward calls to one of a set of destination numbers in round robin fashion at a call center.

Introducing this indirection is conceptually problematic. The 800 number looks like a phone number, i.e. like an E.164 address. But it really isn't an address at all. In fact, it has no location significance. In this sense, it is a *name masquerading as an address*. Surprisingly, this name can be used for routing (up to a point) just like an address. That is, a local exchange, on receiving a toll free number, can forward it to the 800 area exchange just like a 'regular' number. The 800 number, or name, therefore, by masquerading as a regular E.164 address is able to *co-opt* existing forwarding mechanisms to its own ends. The point to take away is that by masquerading names as addresses, we can re-use legacy forwarding mechanisms, yet add flexibility by introducing a level of indirection at any point in the forwarding path.

C. The Internet: masquerading and tunneling

In the early Internet, like the phone network, things were simple. Every interface had an IP address, and the network used routing algorithms to set up forwarding tables, which allowed a router or gateway to determine the output interface corresponding to an IP address. IP addresses therefore were associated with specific locations.

With Mobile IP (and NAT), however, things have changed. With Mobile IP, what looks like an IP address is in fact translated by the home address agent to another IP address, which is (hopefully) the actual physical address. This is essentially the same as what is done for toll free numbers: by masquerading as regular IP addresses, mobile IP addresses, which really have no specific location, can co-opt regular IP forwarding.

In addition to masquerading, the Internet also supports tunneling. Here, a packet has two addresses: an outer address and an inner address, much like an envelope inside an envelope. The network delivers data to destination addressed by the outer address, which uses the inner address to further forward the data.

Masquerading and tunneling both co-opt legacy routing, but differ in one aspect. With masquerading, the final destination is completely specified by the destination address of a packet by means of a translation table (though the translation table can change over time). In contrast, with tunneling, the packet originator is able to exert some control over the final destination of the packet by means of an inner address.

We note in passing that NAT, multicast, and anycast addresses also employ masquerading for their own purposes.

III. Location Specific and Location Independent Identifiers

We have used the term 'address' so far without explicitly defining it. An exact definition raises a conceptual problem. An 800 number or Mobile IP address looks like a 'regular' E.164 or IP address, but it has no location significance, so can we call it an address at all? Probably not.

We argue that much confusion arises from situations where names masquerade as addresses. In order to clarify the situation and distinguish them, we will call them two different things. We call a handle or character string that is associated with a physical location a Location Specific Identifier or LSI. In contrast, we call a handle or character string that is looked up in a table as a Location Independent Identifier or LII. These are similar in intent to Universal Resource Locators (URLs) and Universal Resource Names (URNs) but we have chosen to use other tags because URLs contain DNS names, which are not location specific.

Using these terms, we can call a 'regular' phone number or a 'regular' IP address as LSIs, and an 800 number or a Mobile IP address as LIIs. In the same way, your physical postal address is an LSI, and your name is an LII.

Of course, some LSIs, such as 800 numbers, look syntactically the same as LIIs, but that does not change their essential nature: they are simply masquerading. Because, in this fashion, LIIs can *always* masquerade as LSIs, they are a form of information hiding. Specifically, one part of the network may contract with the other to agree to deliver packets addressed by addresses that appear ostensibly as LSIs. In fact, these addresses may actually be treated by the endpoint as LIIs, and may be resolved to a (putative) LSI.

For instance, one could imagine that the toll free service is provided by a third party. This third party tells the phone company that it has a geographical area that it covers, that happens to be assigned the area code 123. Calls starting with the 123 E.164 address, should therefore be sent to the switch responsible for this area. From the perspective of the rest of the phone network, the 123 area looks like a regular area, and 123 numbers look like regular numbers, i.e. LSIs. In fact they are LIIs. This is how mobile (cell phone) operators can freely interoperate with wireline networks.

Because this information hiding is fundamental, every handoff between service providers can potentially result in an ostensible LSI turning out to be an LII. There is no way to legislate against it or prevent it! Therefore, every identifier must be treated as if it could be an LII unless otherwise proven. We conclude that data delivery may require multiple translations from an LII to a series of one or more masqueraded LIIs finally to an LSI. At this LSI, tunneled LSIs or LIIs can be extracted to make additional forwarding decisions. This process is explained in more detail next.

IV. LII to LSI Translation

By definition, LIIs are not location specific, so it is not possible to deliver data addressed by an LII. In order actually deliver data, is is necessary, at some point, to translate from an LII to a LSI. This requires a lookup in a translation table. Where is the translation table? Its location can be specified in one of three different ways.

A. Explicit pointer to a translation table

In this case, the sender specifies the LSI or LII of the translation table corresponding to a data packet's LII. The destination ID therefore requires two components: the LII to which the packet is destined, and a way to get to the translation table, which is itself either an LSI or an LII.

We now have a problem of recursion. If the translation table's location is specified by an LII, then we need yet another transla-

tion table to get the LSI corresponding to it. Therefore, at some point, the translation must be accomplished using one of the two methods described next. We note in passing that the technique of specifying an explicit pointer to a translation table, though conceptually valid, is not actually used in any system known to us.

B. Implicit determination of translation table

The second technique to translate an LII to an LSI is to implicitly determine the LSI location of the translation table, by convention. The table can be local or non-local. For instance, a web browser implicitly knows that URL LII should be resolved to an LSI (an IP address) using the non-local Domain Name System. The location of the DNS table is found by configuring its LSI on the local host: this LSI is given to the host at the time of IP address assignment using DHCP, or hand-configured in the local file system. Armed with this LSI, the browser can translate the LII to an LSI (which may itself require several indirections).

We now make some observations about this process.

1. the resolution of the LII may result in an LII masquerading as a LSI, such as a mobile IP address or when using call forwarding in the telephone network. In this case, the translation from the masquerading LSI to actual LSI is also done implicitly, using the translation table located at the location indicated by the masquerading LII. For example, one could translate from DNS name (LII) to a mobile IP address (LII masquerading as LSI). When the data gets to the destination specified by the masquerading LII, it is translated to an actual LSI. This translation may also be delegated to an in-line middlebox that promises to deliver data to the LSI, but actually does a translation mid-stream, and generates another LSI. Indeed, this is the way toll free numbers work: there is no need for the call to actually be given to an end system – the middlebox that intercepts the LII implicitly performs the translation.
2. another way to implicitly define a translation table is to use an address schema tag. For instance, strings of the form a.b.c.d are immediately recognized as a FQDN, and sent to DNS for resolution. One can imagine that tags such as I3:GUID can be used to indicate that I3 servers should be used to resolve the LII to an LSI.
3. Another implicit translation mechanism is to use an underlying broadcast mechanism. A node simply broadcasts the LII along with its own LSI and wait for the appropriate node to reply with its LSI. This is how the IP to MAC resolution using ARP works in a LAN; the MAC address is the LSI, and the IP address (LII) is translated to this LSI using ARP. (We have an interesting situation here, in that an IP address is an LSI that can be used for forwarding outside of its own subnet, but within it, while still identifying a physical interface, cannot actually be used for forwarding data!) A MAC address is also an LSI, but it cannot be used for routing (at least scalably) outside of a subnet. Broadcast does not scale, so it can only be used for small networks.
4. LIIs may be explicitly resolved to other LIIs. This allows a considerable degree of flexibility as described in [1].

TABLE I
SOME WELL-KNOWN ADDRESSING SCHEMES

Scheme	*Comment*
Regular telephone number	LSI
Toll-free number	LII masquerading as LSI; implicitly resolved by a middlebox to an LSI
Ethernet address	LSI in the context of a LAN and LII outside it. Can't be resolved outside LAN
Regular IP address	LII resolved by broadcast in the context of a LAN and LSI outside it.
Mobile IP address	LII masquerading as LSI; implicitly resolved by a home address agent to an LSI
Public IP address and port pair	Usually an LSI, but sometimes an LII, masquerading as LSI, implicitly resolved by a NAT middlebox to an LSI
FQDN	LII, implicitly resolved using DNS servers to an LSI
I3 id	LII, implicitly resolved by I3 server (middlebox) to an LSI
HTTP-based URL	LII, resolved using DNS to an LSI or masquerading LSI

C. Default routing

The third way to translate an LII to an LSI is to use default routing. Essentially, the resolver gives up, and simply passes on the LII to someone else, who is better informed, i.e. a default next-hop resolver. This is akin to, but not the same as, default IP routing entries. In default IP routing, the default route is for unknown LSIs; here, default routing is for unknown LII schemas.

Table I explains some well-known schemes in terms of LSI, LIIs, and the translation mechanism.

V. GENERALIZED FORWARDING

Our analysis thus far allows us to construct a generalized forwarding algorithm as follows:

On packet arrival, a node checks if the address is an LII (line 2 below). If so, the LII is resolved to one or more LSIs (3). If the node itself is one of the destinations (6), then it checks if the packet is tunneled (7). If so, the top LSI is popped off, and the packet is re-injected into the forwarding engine (8-9).

If the node is not a destination, the LSI is looked up the next hop interface in the forwarding table 11). If the next hop for the LSI can be found in the forwarding table, then the packet is forwarded on the output interface corresponding to the next hop (13). If the LSI cannot be found in the forwarding table, it is sent on the default next-hop interface for unknown LSIs(15).

If the address is an LII, one of three cases hold. Either it has an LII or LSI of the translation table (18-23), or the node knows (by convention) how to resolve the LII (25-26), or it gives up

and sends the LII to the default next hop resolver(28).

```
1 forward(packet) {
2   if(destination is LII)
3       LSIs = resolve(packet.LII)
4   else
5      LSI = packet.LSI
6   if(this.LSI matches LSI(s))
7       if(tunneled packet)
8            pop(packet.LSI)
9            forward(packet)
10    else
11       lookup LSI(s) in forwarding table
12       if output_interface found
13           send packet(s) on output_interface
14       else
15           send packet to default next hop
16 }

17 resolve(LII){
18    if(explicit table specified for LII)
19       if(table identifier is an LII)
20           table_LSI = resolve(LII.table_LII)
21       else
22    table_LSI = LII.table_LSI
23       return(lookup(table_LSI, LII))
24    else
25      if(LII schema known)
26          return(schema.resolve(LII))
27      else
28          return(LSI of default resolver)
29 }
```

Note that, because forwarding engines are stateful, *every* packet can potentially update the forwarding table state. This fact will be used later to construct dynamic forwarding tables: we have left it out of the description above in the interests of clarity. We have also left out the insertion of a packet into a tunnel, showing only how a tunneled packet can be forwarded. Both dynamic tables are tunnel insertion are show in more detail in the examples in Section VII.

VI. Forwarding primitives

Examining the generalized forwarding algorithm, we note that we can describe the forwarding process with a small set of primitives. We need to be able to lookup LII to LSI translations (line 23) and output interfaces (line 11) in local or remote tables, send packets on an interface (line 15), and, in some cases drop them. We also need ways to update our translation tables to deal with dynamic translation tables, such as those used in NAT. Finally, we need a few helper functions to copy a packet, pop a header (line 8), or create a hash of selected packet headers. This motivates the following set of forwarding primitives.

`d.layer.parameter`: This extracts a parameter from the specified layer in a packet's header.

`d.data`: This refers to the payload of the packet.

send(`d, interface`): This sends the packet called *d* on the interface identified by the second parameter.

drop(`d`:) This drops packet d.

d1 = copy(`d`): This creates a copy of packet d in d1.

h = hash(`value`): This computes a hash of the value using a standard hash function, such as MD5.

`value`= pop(`d`):

push(`d, value`): These remove and add headers to a packet.

`value`= l.lookup(`table, key`):

`value` = g.lookup(`table, key`): This primitive translates from an LII to another LII or an LSI using a table that is either local or non-local (i.e. global).

l.update(`table, key, value`):

g.update(`table, key, value`): These update a local or global mapping from a key to a value. If the key is not already present, it is added to the system. If the key exists, its value is modified. For simplicity, we will assume that keys are very large, so that key uniqueness is trivial. This allows us to update LII to LII or LII to LSI mappings.

VII. Using the Primitives

We now use primitives described in the previous section to succinctly describe some well-known systems.

A. Internet packet forwarding: The base case and MAC broadcast

A packet's output interface is determined by the longest prefix match of the destination address in the local forwarding table. This table has the output interface corresponding to each network number (IP address aggregate), and default route for unknown addresses. If the destination address is the in the same subnet as the node, then a data-link level MAC broadcast needs to be done using ARP to resolve the MAC address of the destination.

We describe the Internet forwarding process as follows:

```
if(d.ip.dst.subnet is this.subnet for this.interface)
    mac = arp_resolve(d.ip.dst, this.interface)
     // details of broadcast elided
    send(d, this.interface)
else
    out_if = l.lookup(fwd_table, d.ip.dst)
     // returns the default out_if if ip.dst is
     // not in the fwd_table
    send(d, out_if)
```

In subsequent descriptions, in the interests of space, we will leave this code fragment, only resolving to the level of an IP destination.

B. Mobile IP: Masquerading and tunneling

Mobile IP uses a combination of masquerading and tunneling. If a Home Address Agent (HAA) recognizes an IP address as a masqueraded address, it resolves this address to the LSI of the care-of-agent, and tunnels the data to the care-of-agent (COA), which delivers the packet to the actual destination.

```
//at HAA, check for masquerading, then tunnel
if(coa_IP = l.lookup(HAA_table, d.ip.dst))
    push(d, coa_IP)
    forward(d)
else //this packet wrongly delivered to the HAA
    drop(d)

//at COA, pop packet destination off
if(this.IP is d.ip.dst)
    pop(d)
    next_IP = d.ip.dst
```

C. ATM: Separate call setup and header translation

In ATM, the call setup packet is forwarded just like a normal datagram. On call accept, the local translation table is updated, and it is used to translate data headers on subsequently arriving cells.

```
if(d.ATM.vci is control VCI) //control pkt
    // elide details of AAL
    if(d.Q.2931.type is SETUP)
        // setup temporary state
    if(d.Q.2931.type is CONNECT)
```

```
        // make temporary state permanent
    out_port = l.lookup(fwd_table, d.Q.2931.dst)
    send(d, next_port)
else
    d.ATM.vci = l.lookup(xlation_table, {d.ATM.vci, in_port})
       // in_port is the port on which the cell arrived
    out_port = l.lookup(xlation_table, {d.ATM.vci, in_port})
    send(d, out_port)
```

D. Email: Global lookup

Email forwarding is done by looking up the destination in the global DNS system which uses one or more MX records to return the DNS names of the responsible servers. These have then to be looked up using A records in DNS to return IP addresses. The code at the originating mail server is the following:

```
dst_servers = g.lookup(dns_mx, d.smtp.dst)
dst_server = highest priority item in dst_servers
dst_IP = g.lookup(dns_a, dst_server)
```

An application level mail gateway follows essentially the same logic, except that the lookups may use tables other than DNS.

E. HTTP Cache: Dynamic forwarding table

To demonstrate dynamic lookup table updates, as well as the use of hashes for lookup, we now describe the actions at a web cache. Essentially, the cache looks up the URL in a GET request in a local table to see if it has the result. If so, the result is returned. Otherwise, the request is sent to the origin server and the reply is both returned and cached.

```
request(d) {
    result = l.lookup(hash(d.http.URL), cache)
    if (result is null)
        out_if = l.lookup(fwd_table, d.ip.dest)
        send(d, out_if) //send to origin
    else
        out_if = l.lookup(fwd_table, d.ip.src)
        send(result, out_if)
}

response(d) {
    l.update(cache, hash(d.HTTP.URL), d.data)
    out_if = l.lookup(fwd_table, d.ip.dst)
    send(result, out_if)
}
```

F. NAT: Dynamic tables and header translation

NAT manipulates packet headers both on the incoming and outgoing paths. When getting a packet from the NATted side, we check if the translation exists. If it doesn't the translation table is updated to map a TCP port to the incoming IP address and incoming source port, otherwise the previously stored assignment is reused. In the other direction, we lookup on destination port and swap values. Note that the code below assumes that we can look up the NAT table using either a hash of the IP/TCP 4-tuple or the incoming TCP port to return either the NAT port or a IP/port pair respectively.

```
global_nat_port = 1025

outgoing(d) {
    if  (nat_port = l.lookup(hash(d.ip.dst, d.tcp.dst_port,
                d.ip.src, d.tcp.src_port)), NAT_table  is null)
                    // not seen this before
        global_nat_port = global_nat_port + 1
                   // update NAT tables
        l.update(NAT_table, hash(d.ip.dst, d.tcp.dst_port,
                            d.ip.src, d.tcp.src_port),
                        {d.ip.src, d.tcp.src_port, global_nat_port})
d.ip.src = this
d.tcp.src_port = global_nat_port
out_if = l.lookup(fwd_table, d.ip.dst)
        send(d, out_if)
    else
        d.ip.src  = this
        d.tcp.src_port = nat_port
        out_if = l.lookup(fwd_table, d.ip.dest)
        send(d, out_if)
}
incoming(d) {
        {ip, port} = l.lookup(NAT_table, d.dst_port)
        d.ip.dst  = ip
        d.tcp.dst_port = port
        out_if = l.lookup(fwd_table, d.ip.dst)
        send(d, out_if)
}
```

G. I3: Using distributed hash tables and combining routing with resolution

In I3, a $trigger$ stored in a distributed hash table (DHT) maps an I3 id (an LII) to one or more I3 ids or one or more IP destinations (putative LSIs). Packets from an I3 endpoint, that may contain a stack of one or more I3 ids, are tunneled to the closest I3 server, which extracts the top I3 id and uses this to forward the packet. An I3 server first checks if the trigger corresponding to this I3 id is stored locally. If so, the trigger is accessed to retrieve either another id or the set of IP destinations. Otherwise, the packet is forwarded to the I3 server that is closer to the server that actually has the mapping using the local $fingertable$. This process is described below, based closely on Figure 3 in [6]. Note that we do not describe tunneling, since the forwarding path is identical to regular IP packet forwarding. Also, note that the finger table lookup shown here converts a g.lookup into an l.lookup, so, in effect, we're showing how to implement g.lookup.

```
i3_recv(d){ // forwarding action for packet d
    id = d.i3stack.top
    if (l.lookup(local_hash_table, id) is null)
         //someone else handles it
        i3_forward(d)
        return
    pop(d.stack)
    set_t = l.lookup(trigger_table, id)
                   // find all matching triggers
    if (set_t is null)
        if (d.i3stack is null)
               // no triggers match; no ids in stack
            drop(d)
            return  // packet is dropped
        else
            i3_forward(d)
            // no triggers but one id still
                 in stack, so use it to forward
     while(set_t != null)
     // forward packets for each matching trigger
         t = set_t.top
         d1 = copy(d) // each trigger creates a copy
         push(d1.hdr.i3stack,  t.id)
          // t's id is pushed on d1's stack
         i3_forward(d1)
}

i3_forward(d){
     id = d.i3stack.top
     if (id is IP address)
        out_if = l.lookup(fwd_table, id)
        send(d, out_if)
    else
        next_IP = l.lookup(finger_table, id);
        out_if = l.lookup(fwd_table, next_IP)
        send(d, out_if)
}
```

DOA [7] is similar to I3 it its use of globally-unique ID stacks, and its ability to resolve UIDs to other UIDs. The resolution algorithm shown above, with small modifications, also describes DOA.

H. Dynamic Source Routing: Flooding for path discovery

In dynamic source routing [2], packet transmission is preceded by flooding, where the control packet header accumulates a route to the destination. The destination uses this accumulated route to reply to the initiator who can then use a source route to reach the destination. The description below does not include the optimizations presented in [2].

```
initiate_flood() {
    for each out_if
        d.dsr.src = this.ip
        d.dsr.dst = destination
        d.dsr.seq = seq++  // set sequence number
        send(d, out_if)
}

incoming_flood_packet(c) {
    // if this flood has been recently seen, drop it
    if (l.lookup(dsr_table, hash(d.dsr.src,
        d.dsr.seq)) is not null)
     drop(d)
    // prevent loops
    if (this.ip is in d.dsr.route)
        drop(d)
    // this node is the destination, so reply
    if (d.dsr.dst is this.ip) {
        d1 = copy(d) //copy the incoming route
        next_hop = d1.hdr.route
        push(d1.hdr.route, this) //add yourself to end
        out_if = l.lookup(fwd_table, next_hop)
        send(d1, out_if)
    } else {
        d.route += this
        for each out_if // flood
           d.dsr.dst = destination
           send(d, out_if)
 }
```

VIII. Related work

The notion of location-dependent and location-independent identifiers is well known in the literature. Reference [7] has a comprehensive survey of past work in the area of location-independent identifiers such as those used in I3 [6] and HIP [4].

Our work is closest in spirit to Delegation Oriented Architecture [7]. Like DOA, we focus in the process of identifier resolution. However, unlike DOA, we do not propose a new location-independent identifier scheme, but, instead categorize existing identifiers as location dependent or location independent, examine masquerading and study the effect of resolution on the forwarding process. Moreover, we decompose the generalized forwarding algorithm into primitives, and the use of these primitives to describe other algorithms. This allows us to propose hardware accelerators that would help not only DOA, but also other forwarding schemes.

IX. Future work

The forwarding primitives described in Section VI can be used in at least three ways. First, they provide single conceptual framework to study forwarding in a number of diverse application areas. This enables the design of much more complex forwarding schemes. Second, using a uniform set of primitives allows schemes developed in one application area to be applied elsewhere. For instance, the use of masquerading as in Mobile IP may be equally applicable in Web caches that provide anonymity. Finally, reducing complex forwarding functionality to a few primitives makes it feasible to contemplate hardware support for these primitives. Unlike past efforts, this hardware can be shared by a large number of forwarding schemes at various levels of the protocol stack. Accelerators can be used for:

1. **Information extraction from the headers.** This can be aided by simple hardware parsers that, on packet receipt, preload registers with appropriate sections of packet headers and hashes of specified fields.
2. **Packet copying.** This can be a heavy overhead for application layer tunnels. Hardware can be used to create a copy of packet as it is received. A pointer to the packet copy, subsequently massaged by an application, can be used to efficiently create tunnels.
3. **Local lookups.** Local lookups can be speeded up using ternary CAMs.
4. **Pushing and popping packet headers.** These can be accelerated either with a dedicated system call or with hardware.

In future work, we plan design hardware accelerators that can be used by a wide range of forwarding schemes.

X. Conclusions

Naming and addressing have long been conceptually problematic. We propose a simple technique to distinguish between them using the concepts of location-specific and location-independent identifiers. A careful consideration of the translation process from location-independent to location-specific identifiers allows us to construct a generalized forwarding algorithm. The generalized forwarding algorithm is composed from a set of forwarding primitives. We show that the same primitives are sufficient to capture a wide range of forwarding schemes. This suggests that acceleration of these primitives could be beneficial to them all.

Our contributions are fourfold. First, we provide a clean conceptual framework to distinguish between names and addresses. Second, we construct a generalized forwarding algorithm that motivates the choice of a set of forwarding primitives. Third, we show how the forwarding primitives can be used to economically describe a variety of forwarding schemes in the literature. Finally, we identify a set of hardware accelerators that can be used by a range of forwarding algorithms.

References

[1] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I.Stoica and M. Walfish "A Layered Naming Architecture for the Internet", Proc. ACM SIGCOMM 2004, Sept. 2004.

[2] D. B. Johnson, D. A. Maltz, and J. Broch. "DSR: The Dynamic Source Routing Protocol for Multi-Hop Wireless Ad Hoc Networks." in Ad Hoc Networking, edited by Charles E. Perkins, Chapter 5, pp. 139-172, Addison-Wesley, 2001.

[3] P. Mockapetris and K.J. Dunlap, "Development of the Domain Name System, Proc. ACM SIGCOMM 1988, Stanford, August 1988.

[4] R. Moskowitz and P. Nikander "Host Identity Protocol Architecture," draft-moskowitz-hip-arch-05 IETF draft (work in progress), Sep. 2003.

[5] J.F.Shoch "Inter-network Naming, Addressing, and Routing," 17th IEEE Computer Society Conference, Washington DC Sept. 1978.

[6] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, S. Surana, "Internet Indirection Infrastructure," Proceedings of ACM SIGCOMM, August, 2002.

[7] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, S. Shenker, "Middleboxes No Longer Considered Harmful," 6th Usenix OSDI, San Francisco, CA, December 2004.