# Who needs addresses?

V. Guruprasad
*prasad@us.ibm.com*
IBM T J Watson Research Center,
Yorktown Heights, NY 10598, USA.

**Abstract**

I describe a framework for virtualising network addresses, reasoning that virtualisation requires reversing the usual the network-first, resource-second addressing order. A hierarchical name space is constructed for representing the shared resources by a homogenised form of URLs, a logical end-to-end route is constructed by combining the bind and connect requests over this name space, and used instead of end-to-end addresses for setting up virtual paths. Direct interpretation of the homogenised URLs ensures logical routing performance comparable to IP routing, and the logical route provides geographical trimming of the search in the transation to virtual paths. Resulting features include a simple, uniform API for both point-to-point and multipoint connectivity, host security by invisibility, etc., which can be availed by privately deploying the framework over the existing networks.

## 1 Overview

Described below is a framework for virtualising network addresses, which is operable over existing networks and promises novel features at adequate performance. The virtualisation fundamentally solves the address space bounds that exist in current networking technologies, including the Internet Protocol (IP) and Asynchronous Mode Transfer (ATM). It is an internetworking solution in that it enables applications to connect across any combination of networks, which differs from the virtual network approach of IP [1], in that IP addresses identify physical hosts and are therefore real. Virtual network addressing means that the applications should not have to, or be able to, identify physical destinations at all, which evidently cannot be achieved by any scheme of address translations alone. Clearly, the very need for destination addressing at the application layer needs to be overcome, and the question is, is this at all possible, and at a reasonable cost?

Our answer is yes, and it involves inverting the roles of network and local resource addresses, as follows. In discussions of networking, it is natural to represent the network by a graph $\Gamma$, and to number its vertices for reference, say $n_i$, $i = 1...N$; the index $i$, which identifies the node, is then its real address. A virtual address space would seem to be obtained by a mapping $V(\Gamma)$, $\Gamma \equiv \{n_i\}$, but this is insufficient if the $V(n_i)$ continue to identify physical hosts, as is the case with IP, which is such a mapping. Our solution involves supporting a multitude of coexisting mappings $V_j(\Gamma)$, $\Gamma = \{n_i\}$, each $V_j$ being one-to-one. If we were to associate one $V_j$ at each node $n_j$ in the network, we would arrive at "source-dependent addresses", of which ATM virtual path indices and UUCP are well known examples, and we have already learnt from the latter that source-dependent addressing cannot lead to an application or user-friendly solution, which is why we have end-to-end addressing in ATM. In both native ATM and IP networks, however, an application typically has to first look up a Domain Name Service (DNS) for an address to the destination host bearing the desired resource, and we know from ATM that translating addresses to routes is an NP complete problem [2], *unless* we could directly interpret the address, as in IP routing.

We accordingly route-interpret the resource locator itself, instead of dereferencing it in the DNS for an address, thus identifying the mappings $V_j$ with shared resources instead of hosts, so that the total address space $\Xi \times \Gamma$, $\Xi \equiv \{V_j\}$, becomes inherently unbounded, $V_j(\Gamma)$ being interpreted as connections to that resource. We use a DNS-like service to represent resources, and interpret modified universal resource locators (URLs) over it to obtain end-to-end logical routes, and use the logical routes to set up virtual paths, avoiding end-to-end addressing at all levels.

I explain my purpose and motivation in the next section, followed by the basic scheme of the framework and its principle of operation in §3, and discuss its expected performance in §4. ACS availability, route discovery and incremental translation are considered in §5, and the security aspects of virtualisation in §6. Lastly, I describe the prototype system calls API in §7, which illustrates the inverted form of the virtual addressing mentioned above, and the current status of implementation, in §8.

## 2 Motivation and scope

Our concern is partly academic, as a cross-check on the fundamental assumptions that would be otherwise taken for granted. Our result shows that our current embracement of IP is not as well thought out as it should be, and its original principles are in any case being gradually eroded, prompting a reconsideration: in routing, by multi-network label-switching (MNLS) and the use of network address translation (NAT) for virtual private networks (VPNs); and the current usage of the Internet, where URLs have emerged as the *de facto* form of addresses, questioning the very need for the visibility of IP addresses. Both issues are exploited in our solution, as we no longer depend on the eroded end-to-end significance of IP addresses, and apply IP's route-interpretation principle directly to the URLs albeit in a slightly altered syntax.

We also have a parallel in the operating system (OS) world, where lesser OS's that ran applications with real memory addressing, were adequate for popular consumption until quite recently, when the sophistication of the Internet caught up. It leaves us no reason to believe that a similar switch to virtual addressing in the networking field would be for ever unnecessary. It is also currently overlooked that the consumption of addressing resources is not linear but exponential, and there is no reason for the 128-bit space of IPv6 to last more decades than the original 32-bit IPv4 space, unless we hold back for some reason.

Several clarifications appear to be in order. First of all, our framework does not force us to advocate ripping out the existing IP infrastructure, although it could be operated as efficiently without IP. Rather, IP, like other existing technologies including Asynchronous Transfer Mode (ATM), would be readily availed of as transport, as demonstrated in our prototype (see §7). Moreover, some form of real addressing would be necessary for configuring the framework, and IP, if available, would be simply more convenient in the local administrative zones. This does not diminish the role of the framework as IP would not be used in its original, end-to-end role.

It does, however, obviate the need for the Internet *per se* to migrate to IPv6 after all, which would impact the addressing of Web sites, for example, and would allow emerging wireless networks to avoid needing a share of the IP space, the demand being doubled by the Mobile IP protocol (RFC 2002) which requires two addresses for each mobile device. Furthermore, wireless devices should not need IP, and ATM networks, both wired or wireless, IP or Local Area Network (LAN) emulation, merely for connectivity with the rest of internetworked world, which is the case at present. There is also potential for simplification in the administration of address blocks, where dynamic allocation is inappropriate.

We employ virtual paths because the technique already incorporates NAT at the switching granularity, so that an individual virtual path can extend across an indefinite number of IP spaces. Note that this does not contradict our previous observation that a product of translations alone cannot provide virtualisation, as the path indices are obtained only as a result of the virtual-to-real translation of the logical route by the ACS. The translation may be viewed as a distributed form of on-demand compilation of the application-supplied pathnames, the objects compiled being the virtual paths. The network configuration data used for the translation is correspondingly representable as an attribute grammar, the attributes representing traffic, quality of service and possibly authentication parameters, and an analogous case exists for one-time interpretation, which would avoid the virtual path translation and signalling, as described in §3.

Third, there appears to be some degree of overlap in functionality with existing networks. Our distributed name service, which we call Addressless Connection Service (ACS) because it does not return addresses and is itself responsible for logical routing, as mentioned, seems to overlap the DNS in applications enabled with our application programming interface (API), but this is superficial, as the DNS would have been used to configure the framework. Overlap seems to exist also with respect to signalling, when used over ATM, for example, which itself provides virtual paths, but as the particular form of broadside-on signalling we need is not generally available to us, we would treat the ATM fabric as an opaque cloud, also avoiding having to deal with routing within the fabric in the process.

Fourth, although the ACS looks a lot like the DNS, our pathnames represent application contexts, not the hosts from which they were defined. In the Universal Resource Locators (URLs) used on the Web, differences between the DNS and the file system conventions make the network address and the host resource parts readily distinguishable, but would be inappropriate in our context, hence we prefer a homogenised form of the URLs, or HURLs, which we literally interpret against the ACS hierarchy like a file system pathname. This keeps the logical routing complexity linear, like IP routing, which interprets addresses similarly, and the burden of search lies wholly in the translation to virtual paths. Since this is partitioned along the logical routes, the ACS servers are intermediate between DNS servers, which do not do routing, and the ATM switches, which are individually responsible for end-to-end routing. In some envisaged applications, like providing business-to-business (B2B) services across private LANs (§6), we can hope for a linear routing cost because the search would be limited to a narrow corridor along the logical route for switches located at the gateways, which we shall refer to as the *geographic principle*. Note that the principle actually holds better for wide area networks (WANs) than to local area networks (LANs) because WANs have fewer interconnections, i.e. less geographic freedom in network terms.

Our method has the advantage of using *a priori* end-to-end directivity over both source and hop-by-hop routing methods, in so far as best effort routing is concerned; we would be much slower if backtracking were involved because of the distributed nature of the virtual path translation. Since our primary purpose is virtualisation and not networking *per se*, we would be satisfied with obtaining this at reasonable cost over existing infrastructure and with independent, private deployment as needed. An incremental strategy is also described in §4 to exploit the geographic trimming for dynamic path repair, and might compare favourably with rehealing methods known in ATM [3] [4].

## 3 Addressless principle

The principle underlying the present framework is the observation that a combination of `bind` and `connect` operations is both necessary and sufficient for logical connectivity between any pair of processes: necessary because the two processes must actively cooperate, and sufficient because at least one of these operations can stretch across the network, as we know from the `socket` API. The problem has been that since `bind` was defined to be local, the `connect` had to be stretched all the way to the binding host, requiring the latter to bear an address visible end-to-end and preferably advertised over a name service like the DNS.

The only way to avoid end-to-end addressing, then, is to redistribute some of the work over to the `bind` operation, making both processes reach out part of the way and make the logical connection in the middle. We would then have traced out an end-to-end logical route, and since this is already end-to-end, we should not need end-to-end addressing for transferring short messages, or for setting up end-to-end data streams, the latter depending on the geographic principle for efficiency.

A sample ACS configuration is illustrated in Fig. 1, comprising a hierarchy of connection servers $A...J$, to serve application processes running on hosts $a$, $b$ and $c$ via the switches $p$, $q$, $r$ and $s$. The logical hierarchy is represented by solid arrows; servers $B$ and $F$ are shown linked both ways, to illustrate the likely scenario where the root of the hierarchy is conceptual and not embodied as a server, as in the Andrews File System (AFS). Each connection server is typically linked with one or more switches, as shown by the broken two-way arrows, for signalling, and the virtual paths are set up over the transport links, shown as bold lines, between the switches and the hosts. Note that the connection servers and the service links linking them to the hosts and the switches are both conceptual, and could be physically embodied in various ways. Our illustrative example will be a *client-server* application, wherein the application client is a process $v$ executing on host $b$ that seeks connection to the application server process $u$ on host $a$.

We shall designate the application service context by a HURL `//F/G/I/x`. By the foregoing principles, process $u$ must first *advertise* its service by invoking `bind("//F/G/I/x")`, and the client process $v$ then makes a request `connect("//F/G/I/x")`, i.e. using the same HURL, which would have been bound to a name $x$ on the connection server $I$, regardless of the caller's location in the network. Its interpretation in the ACS is straightforward: connection server $A$, on receiving the `bind` request from process $u$ on host $a$, finds that `//F` is not on its direct line of descent from the root, and passes the HURL intact to its parent, $B$, which, knowing that it has no parent, looks for a peer server named $F$. The request then descends from $F$ through $G$ to $I$, and since $x$ is the last component identifier in the HURL, $I$ binds it locally to a newly allocated context, in which it conceptually stores the partial logical route **xIGFBAau**, which is the reverse

of the route traversed by $u$'s request.

When the `connect` request from process $v$ reaches $I$, it will have similarly traced out the partial logical route **vbEFGIx**, assuming that the request went to server $E$, and not $H$. $I$ then concatenates the two partial routes to obtain the end-to-end logical path **vbEFGIxIGFBAau**, and optimises it by discarding the repeated segment **FGIx**, to obtain the shorter path **vbEFBAau**, which is then used for signalling the virtual path, as will be soon explained.

We first consider the case where the client requests sending of a one-time message to the server process $u$, for example in a paging system. All that $I$ needs to do is to pass it onward to $u$ backward along the previously registered path **xIGFBAau**. There is no need to store the second partial route **vbEFGIx** unless a reply is expected, and it can be discarded after forwarding the reply. The reverse route **IGFBAau** is also conceptual, for we could just as easily compute the reverse HURL `//B/A/u/a` to bind with the name $x$ at server $I$, which is more likely in the typical implementation because the HURL representation is compact and can be as efficiently interpreted as its UUCP-like path expansion **xIGFBAau**. We also have an obvious opportunity for optimising the paging path by eliminating the repeated segment **FGIx**, which requires the context information to be percolated upward to server $F$. Since the latter is higher in the hierarchy, it is important not to load it unnecessarily, for which a caching strategy seems to be most appropriate.

The complete logical path **vbEFBAau** is involved in constructing a separate virtual path, say **uapqsbv** or **uapqrsbv**, for subsequent data. Fig. 2 illustrates the suspension-bridge relation between the logical and virtual paths, together with the switch path table entries comprising the latter. Each entry is shown as a tuple $\langle \sigma \cdot \pi \rangle$, where $\sigma$ identifies outgoing next-hop link, $\pi$ is onward path index; both are expected to be small integers in practice. Also shown are the outgoing file table entries within the operating systems on the two end hosts; the incoming paths presumably lead to the same entries, because the same file structure is typically used for traffic in both directions.

The virtual path is obtained as follows. As shown in Fig. 1, host $s$ is linked to two servers, $E$ and $H$, by configuration. It first sets up a tentative file table entry, reachable by return index $\theta_2$, and requests server $E$, as assumed above, by invoking `connect("//F/G/I/x", `$\theta_2$`)`. Server $E$ first translates the HURL to the triple $FEb$ by reference to the ACS. The switch information is conceptually configured as context-sensitive translation rules of the form

$$F \cdot E \cdot b \mapsto F \cdot s \cdot b \quad \text{\# Note: hosts} \equiv \text{switches}$$

treating the current (middle) server id as nonterminal and the switch ids as terminals in the production. Our sample rule says that switch $s$ may be applied to paths between $F$ and $b$. The idea lends itself to generalisation with attributes to represent cost, bandwidth, latency, etc., and accommodation of several such rules at each server for each pair of its neighbours, accepting similar parameters with the the `bind` and `connect` requests in order to pick the best match for each connection.

Assuming the switch selected is $s$, $E$ then signals $s$ to allocate two entries in its path table, supplying the return index $\theta_2$ and the requestor's identity $b$ as the return destination as arguments. Note that the identity of $b$ needs to be unique only with respect to the immediate switch $s$, and is expected not to be a universal address. In practice, $E$ could be configured to transform $b$ to a link identifier with respect to $s$, using configured information once again, or pass its own link identifier for $b$ and depend on $s$ to perform the corresponding translation. In any case, $s$ enters these arguments into the first of the allocated entries and returns indices to both, $\theta_1$ and $\delta_2$, to $E$. $E$ now passes the request onward to the identified next server on the logical route, $F$, together with the HURL and the new return index $\delta_2$. $F$ does a similar translation and signalling to the next switch $r$, except that the return destination is now switch $s$ instead of server $E$, for obvious reasons. The sequence continues until the request reaches host $a$ bearing the return index $\alpha_2$, where process $u$ would be typically sleeping on an `accept` operation. If it honours the request, a new file table entry is created within the OS on host $a$, bearing the return index $\alpha_2$ and destination $p$, and a new index $\alpha_1$, leading to this file table entry, is returned in the response from $a$ to the last server $A$ in the logical path.

$A$ now signals to switch $p$ once again, telling it the forward index $\alpha_1$ and destination $a$ to complete its previously allocated second entry, and in turn passes back to server $B$ the value $\beta_1$, previously returned by switch $p$, as the forward index, along with the forward destination identity $p$, to pass to its switch $q$. This return sequence continues until $E$ receives the forward index $\delta_1$ and destination $r$ from $F$, which it promptly passes to $s$ once again, before returning the forward index $\theta_1$ and destination identity $s$ to host $b$, to be entered into its previously allocated file table entry as shown. The complete sequence is summarised below,

showing the intermediate logical path fragments and the sample state information held at server $E$:

1. $b \to E$: `makepath(`$\langle b \cdot \theta_2 \rangle$`,` **FBAau**`)`.
   $[E{::}\ b{:}\theta_2]$

2. $E \to s$: `makelink(`$\langle b \cdot \theta_2 \rangle$`)`;
   $[E{::}\ b{:}\theta_2\ \ s]$

   $s \to E$: `return newlink(`$\theta_1,\ \delta_2$`)`;
   $[E{::}\ b{:}\theta_2\ \ s{:}\theta_1{,}\delta_2]$

   $E \to F$: `makepath(`$\langle s \cdot \delta_2 \rangle$`,` **BAau**`)`.
   $[E{::}\ b{:}\theta_2\ \ s{:}\theta_1{,}\delta_2\ \ F\ ]$

3. $F \to r$: `makelink(`$\langle s \cdot \delta_2 \rangle$`)`;

   $r \to F$: `return newlink(`$\delta_1,\ \gamma_2$`)`;

   $F \to B$: `makepath(`$\langle r \cdot \gamma_2 \rangle$`,` **Aau**`)`.

4. $B \to q$: `makelink(`$\langle r \cdot \gamma_2 \rangle$`)`;

   $q \to B$: `return newlink(`$\gamma_1,\ \beta_2$`)`;

   $B \to A$: `makepath(`$\langle q \cdot \beta_2 \rangle$`,` **au**`)`.

5. $A \to p$: `makelink(`$\langle q \cdot \beta_2 \rangle$`)`;

   $p \to A$: `return newlink(`$\beta_1,\ \alpha_2$`)`;

   $A \to a$: `makepath(`$\langle p \cdot \alpha_2 \rangle$`,` **u**`)`.

6. $a \to A$: `return linkage(`$\langle a \cdot \alpha_1 \rangle$`)`;

   $A \to p$: `setlink(`$\beta_1,\ \langle a \cdot \alpha_1 \rangle$`)`.

7. $A \to B$: `return linkage(`$\langle p \cdot \beta_1 \rangle$`)`;

   $B \to q$: `setlink(`$\gamma_1,\ \langle p \cdot \beta_1 \rangle$`)`.

8. $B \to F$: `return linkage(`$\langle q \cdot \gamma_1 \rangle$`)`;

   $F \to r$: `setlink(`$\delta_1,\ \langle q \cdot \gamma_1 \rangle$`)`.

9. $F \to E$: `return linkage(`$\langle r \cdot \delta_1 \rangle$`)`;
   $[E{::}\ b{:}\theta_2\ \ s{:}\theta_1{,}\delta_2\ \ F{:}r{,}\delta_1\ ]$

   $E \to s$: `setlink(`$\theta_1,\ \langle r \cdot \delta_1 \rangle$`)`.

10. $E \to b$: `return linkage(`$\langle s \cdot \theta_1 \rangle$`)`.

The above example illustrates the setting up of a two-way connection, and shows that each switch is signalled to twice, once for each direction. The same procedure would be executed for a one-way connection, except that the switches would be signalled to only once, in the desired direction.

# 4 Performance

The cost of computing the virtual path is distributed over the logical path but is clearly cumulative in terms of delay. We also incur a signalling delay proportional to $2n + 4m$, where $n$ and $m$ denote the number of hops in the logical and the virtual paths, respectively. This reduces to $2n + 2m$ for one-way paths, $2n$ for paging with reply, and to just $n$ for paging without reply. The last case corresponds to datagram routing, and we have a slight advantage over IP in avoiding the DNS lookup. In general, however, these delays make the framework slower to respond to changes than IP, but recall that our method is particularly intended for crossing IP's address boundaries, where the additional delay would be acceptable. For instance, the SOCKS protocol requires a signalling TCP connection to be first made with the gateway; in our method, the service links would have been already set up, avoiding the per-connection delay. Both $n$ and $m$ are also expected to be small numbers in such applications.

Our translation complexity also compares favourably against that of source routing, which is prescribed by the ATM private network-to-network interface (PNNI) specification [5,§3.7], and involves an expanding search from the requesting host, the search space growing as $b^m$, where $b$ denotes the mean branching factor and $m$, the number of hops from the source. The complexity is reduced by the hierarchical clustering, but the result would be no match for direct interpretation, like IP routing, if it were available. Our method reduces the search space to $nb^2$, making it comparable to hop-by-hop routing, which is not preferred because of the likelihood of loops. We are immune to looping because each incremental translation is pinned at either end to the logical path, and not open-ended as in hop-by-hop routing. In our example above, server $E$ begins with the triple $FEb$, and seeks to replace itself therein with a suitable switch. In hop-by-hop routing, host $b$ would have supplied $a$'s address, from which $E$ cannot *a priori* determine a next logical point $F$ to direct the hop, so that the search diverges from the source $b$ and can easily loop back.

These estimates are purely theoretical, and for the best case scenario. The worst case is when multiple paths are possible and the most optimal, according to some criterion, is to be found, involving backtracking over the search space. Although the total search space is bounded in our framework by the logical path, any backtracking along it would incur additional signalling delays. We are thus more strongly limited to first

fit and best effort routing, and are dependent on the geographic principle for efficiency.

It should be observed that relaxing the geographic principle allows for more optimal routes and the *vice versa*, so that the geographic freedom determines the tradeoff between "compile-time" or path set up latency and the "run-time" optimality of the virtual paths.

## 5 Recovery and incremental routing

There is scope in the framework for route discovery, for example, server $F$ could cache the subpath **FGIx**, on recognising repetition of references to `//F/G/I/x`, and use it to forward paging messages directly from $E$ to $B$ instead of via $G$ and $I$, reducing the load on itself. More important would be the discovery of switches, to be reflected to the translation rulebase at the nearby connection servers, in accordance with the geographic principle. Multiswitch and multiserver rules are easily accomodated in approach, e.g.

$$\begin{aligned} B \cdot F \cdot E &\mapsto B \cdot q \cdot s \cdot E \\ B \cdot F \cdot E \cdot b &\mapsto B \cdot r \cdot b \end{aligned}$$

which could be applied in various ways, for instance, nondeterministically, by priority based selection, etc. These generalisations are still being investigated.

In general, however, route discovery is meaningful only within a given network, as having an automaton transcend gateways and firewalls on its own would be quite undesirable. It is only within the individual hops, therefore, that we would expect discovery to be useful, but at this level, it would be already performed within the underlying transport network in our envisaged deployment over existing networks. The only reason for having multiple hops within an IP network would be security, as described in §6.

In the context of recovery, it should be noted that the reliability and availability the connection servers can be readily assured by setting up multiple servers at each node in the ACS tree, using known techniques, for instance those used in high end web servers, for caching and load balancing. The burden of state on our connection servers thus does not seem to be a critical problem, but more study is needed to determine how long the state is best retained, and whether it would be optimal to have timeouts or cache-like replacement, because the state would be useful as scaffolding for rerouting virtual paths when a link fails. This would, of course, require the switches to signal back to the connection servers.

Alternatively, the state can be stored at the switches, thereby freeing the connection servers, as the state data would be of the same order of size as the path table entry. This is because it would be distributed among the servers or switches along the respective paths. The burden thus does not appear to be substantial and the choice of location would depend more on convenience.

For instance, in our example system, switches $q$ and $s$ would have no difficulty rerouting paths via each other if link $(rs)$ failed, if discovery were enabled between switches. Instead, consider how we could do the same using our framework incrementally without discovery. Recall that in the original signalling sequence, servers $B$, $F$ and $E$ would have performed the translations $ABF \mapsto AqF$, $BFE \mapsto BrE$ and $FEb \mapsto Fsb$, respectively, which is how we would have got our path through $r$ in the first place (Fig. 1). When $r$ or one of its links fails, it is as if $r$ were suddenly missing from the path, and the incremental translation should ideally produce the same result as if $r$ were never present. $F$ would have been unable to replace itself in the logical path with $r$, and instead, we would have had larger logical segments $ABFE$ and $BFEb$ to translate at $B$ and $E$, respectively.

We would have had to map $s$, identified by $E$, as the next hop to $q$, which is signalled to by $B$, using $F$ as intermediary in the translation, and similarly for $q$ as the next hop for $s$ in the signalling by $E$. There are only two ways to achieve this translation without breaking the premise of end-to-end addressless: introduce a link id tuple $\langle q' \cdot s' \rangle$ in the configuration supplied to $F$ say, so that id $s'$ as known to $q$ and id $q'$ as set up at $s$, both refer lead to the link $(qs)$, or define a local address space for just the three switches $q$, $r$ and $s$, such that the signalled ids will be interpreted by these switches as addresses in this special address space. In either case, when the breakage is detected, switches $q$ and $s$ need to notify their respective servers $F$ and $E$. In order to rekindle the incremental computations, the original triples $ABF$, $BFE$ and $FEb$, would have to be either saved and available within the servers, or copied to and saved by the switches for this purpose. We expect this incremental translation to perform better than an adoption of rehealing techniques from ATM, for the reasons explained in §4 for end-to-end routing.

## 6 Firewalls and security

An obvious feature in our framework is the "security by anonymity" resulting from the concealment of host

identities even from the connecting processes, because the HURLs bear no correspondence whatsoever to the hosts of the application servers advertising them on the ACS. A client can extract route table entry within its OS corresponding to a connection in our framework, but the address contained therein can only lead to the first switch on the virtual path. There is no incentive for the client to mount a denial of service attack on that address, because that would only cost the client's own connectivity to the network beyond the switch, and the attack cannot be initiated from a different location in the network because the virtual path indices are source-dependent. To compare, SYN attacks have been mounted in IP precisely because the victim is exposed by its address and can be attacked from other locations on the Internet so that the attacker does not get cutoff by choking of routers in its own neighbourhood.

In our framework, the victim is exposed only within its local IP space, and it is easy to see that the isolation is proportional to the number of hops in the virtual path between it and the attacker, assuming that the attacker is able to acquire a connection in the first place. This is similar to the protection in the Onion Routing (see `http://www.onion-router.net`) for data. The invisibility of host addresses is at first daunting, as it would seem to block administrators just as surely as adversaries. Legitimate administrative access is not really blocked, however, as local network addressing, including IP, remains available under the framework, and even remote administrative access could be supported, where needed, by advertising an SNMP-like service over the ACS, although this would inevitably diminish the security to some extent. What distinguishes our framework is primarily that it is not promiscuous and access of any kind must be set up actively, including those compromising security.

Our connection servers would be similarly protected by selfishness, as it were, because an attack on the ACS would choke the very server used by the attacking host; again, a given connection server would be vulnerable within its own IP network, and the protection increases with more levels in the hierarchy. This is of concern because unlike the DNS, the ACS is not a directory service, but instead controls access to data and services.

The separation between HURLs and their originating hosts permits an even stronger form of protection, by authenticating connect requests within the ACS and away from the application server hosts. This could also be applied to bind requests, to restrict advertisements from foreign hosts on private connection servers, for business-to-business services in the private space, which in turn could use connect-time authentication to ensure licensed usage. These ideas are illustrated in our example by the three firewalls $X$, $Y$ and $Z$, shown by thick broken lines in Fig. 1. $Z$ lies behind $Y$ and hides host $c$ and servers $G$, $I$ and $J$ from host $b$ and server $E$, which are themselves hidden behind $Y$ from servers $B$ and $C$. The notion of firewalls makes sense only in terms of the local network address spaces to which these nodes belong. Thus, $r$ and $c$ cannot share a common IP address space, and the links from $c$ and $G$ to $F$ would have pass through a SOCKS or NAT gateway. Our capability for remote advertisement allows the application process $u$ on the foreign host $a$, protected behind its own firewall $X$, to advertise on $I$ behind two layers of protection, provided it has been given the HURL to use, `//F/G/I/x`, and the keys to authenticate itself to the servers $F$, $G$ or $I$ that may challenge it.

## 7 The system calls

As mentioned in §1, the virtualisation depends on a reversal of order in the addressing of shared resources from the usual *host* × *resource* form, common to the Internet URLs and almost all existing APIs concerning shared resources, including the Network File System (NFS), the Network Information Services (NIS) and the Distributed Inter Process Communication (DIPC), to the inverted form *resource* × *virtual address*, meaning that the resource itself must be addressed first, as if it were mirrored on every host, with the virtual address denoting indexing within the resource, but of a network kind, which would be inappropriate, for instance, to addressing within a shared memory region. Rather, the virtual address has a natural interpretation in a messaging context as the ordering of network connections made in it, like the "communicators" in the Message Passing Interface (MPI) library, each of which constitutes an array of peer end points that could be likened to file descriptors.

Since we could bind multiple HURLs for a given resource, the ordering cannot be handled by the ACS, but must be managed by the applications themselves. The issue is of concern only in multipoint connections, as in a typical server application, a natural ordering is trivially obtained by counting the client connections in the order they were made. Accordingly, it suffices to have a hook within the API to allow applications to implement their own orderings if needed, and the only necessity is support for multiple message end points.

These considerations are the basis of our prototype API, wherein we represent a messaging end point by the two-dimensional address ⟨`cid` · `vaddr`⟩, where `cid` identifies the context and `vaddr`, a connection number within that context. The `cid` is encapsulated in the OS by a *cnode* data structure leading to a virtual path table indexed by `vaddr`. Each cnode is then a local image of a context, and as it represents a shared resource in any case, it seemed natural to give it an independent existence like a System V IPC id, replete with user and group ownerships and *rwx* permission mode bits, so that it could be set up and used for local IPC even without binding a HURL.

A new system call, `context(type,mode,arg)`, was defined to create the cnode structure within the kernel, and to return a local context id. A second system call, `cntl(cmd,arg)`, was defined in the style of `ioctl` for performing various operations on the cnodes, including their deletion, and a third, `cbind(cnode,hurl,arg)`, for binding a HURL to a cnode. The ACS lookup call `cget(hurl,type,mode,arg)` looks more complicated because it generally returns a new cnode on success, in order to support multipoint usage. The existing `read/write` family of system calls was reused for messaging I/O by instantiating file descriptors for each local connection, the extra argument in the POSIX `readx/writex` calls being used for specifying the peer `vaddr`. Note that we have no room for an equivalent of the socket `send/receive` system calls, as the only extensions these make over `read/write` are for destination addressing.

A fifth call, `copen(cnode,oflags,arg)`, was defined for opening local connections only, as opens on peer hosts are expected to be transparently reflected to the local cnode by a synchronisation protocol in the `type`-implementation code. In the switch table entry for the sample TCP-server type, the `copen` field pointed to the internal entry point of the socket `accept` call. The corresponding redirect of `copen` to the socket `connect` in the sample TCP-client type switch entry would have been wasteful, however, as it would have required client applications to call the kernel twice, with `cget` and then `copen`. The `copen` hook therefore made a `noop` and the `cget` was overloaded to also call `connect` and to return the resulting file descriptor instead of the context id. Brief code examples are given in Appendix A to illustrate the API. A symmetric multipoint type was successfully implemented using UDP and a prototype synchronisation protocol for the `vaddr` ordering, primarily as a test of the system calls and switch table design.

Very simple name servers were written in Perl and C for exercising the prototype API code. Communication with the name servers was managed by a user-mode `cxd` demon process via a special device driver. While the API has sufficed for studying the host implications, the investigation was originally concerned with an elegant design for MPI support in hardware on the IBM SP; the above design particularly permits lazy switching of the virtual path tables in the interface. A command interface was also provided encapsulating calls to `cntl` and `ioctl`, on cnodes and file descriptors, respectively, and a `/proc` file system implementation was examined.

## 8 Work in progress

Many networking tools have been built to study the issues involved in bridging firewalls and networks in the course of various projects. Some were included in the IBM RTSP reference toolkit, and were used to mirror the development server site to the Internet. One is in active use proxying our Deep Blue game server on the Internet, whose design was a result of these experiments, and which uses RTSP for session control. A special device driver interface was designed for using ATM AAL5 virtual paths from the command line, for eventual use in scripting the ACS configuration, as well as a prototype switch for deploying the framework over IP. Prototypes of the ACS and the path translation are still being developed. Also being examined is a very low latency signalling protocol that builds on the present framework, and would incorporate a notion of caching.

## 9 Conclusion

I have presented a notion of virtual network addressing patterned after the per-process virtual address spaces available for memory addresses in modern operating systems, that would have a number of useful features, including host security by invisibility, Web-like access as its primary means of connection, and perhaps most importantly, elimination of real network addresses and address space issues from the application and user interfaces. I have argued that it can be deployed over existing networks and the Internet, giving reasons why it would perform adequately for this purpose, viz that it would essentially interpret user or application-supplied URLs for routing, like the interpretation of addresses in traditional IP routing, instead of divergent search from the caller's host, as in ATM. I have also shown that it leads to small API resembling but simpler than the socket API, and at once more capable, in that it

would uniformly support multipoint connectivity for distributed parallel applications.

How the framework would be used on a larger scale has not been described, nor examined in detail. Also missing is a better understanding of the geographic principle, i.e. the impact on the route translation from the end-to-end directivity available in our approach. The B2B opportunity needs to be explored, as well as the possibility and impact of geographical localisation of ACS advertisements. We hope to explore these issues once we have a full prototype working. We have not explored application of these ideas to telephony, where the 9-digit number space is similarly getting filled up.

## Acknowledgments

To Nick Trio for coining the term *addressless* and much needed initial encouragement, And to the many friends and colleagues whose time and advice have influenced and guided this work.

## References

[1] D Comer. *Internetworking with TCP/IP*. Addison Wesley, 3rd edition, 1995.

[2] R Izmailov, A Iwata, and B Sengupta. ATM routing algorithms for multimedia traffic in Private ATM Networks. *Jour. Heuristics*, 6(1):21–38, Apr 2000.

[3] P Georgatsos et al. Technology Interoperation in ATM networks: the REFORM system. *IEEE Communications Magazine*, 37(5):112–118, May 1999.

[4] Y F Wang and R F Chang. Self-healing on ATM Multicast Tree. *IEEE Trans. Commun.*, E81-B(8), Aug 1998.

[5] *ATM Private Network- Network Interface Specification version 1.0*. The ATM Forum, http://www.atmforum.org.

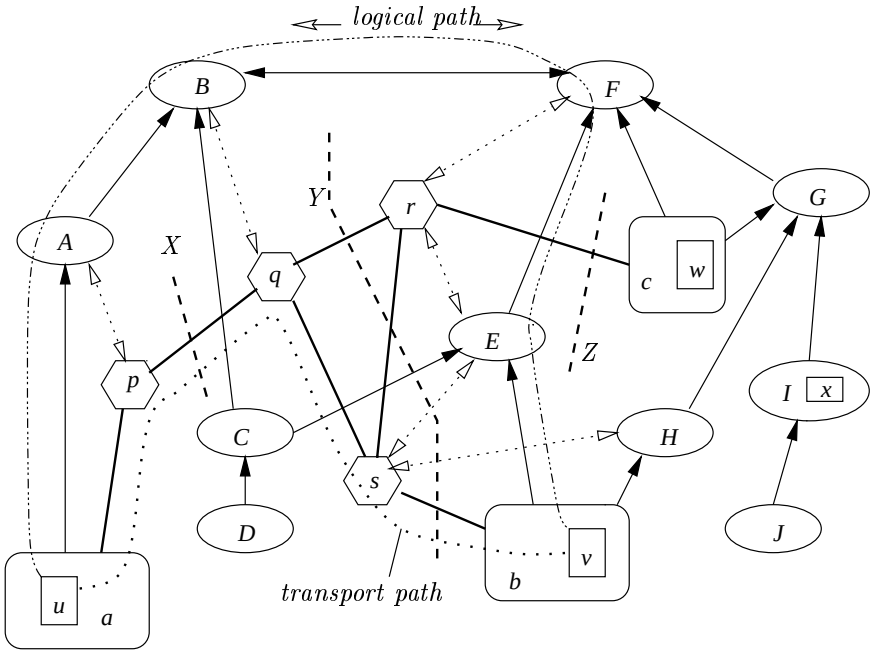


Fig. 1: Logical and physical paths

## A Programming examples

```
1   // ------------------------------------------------------------------
2   // a simple stateless server a la HTTP/1.0:
3   // ------------------------------------------------------------------
```

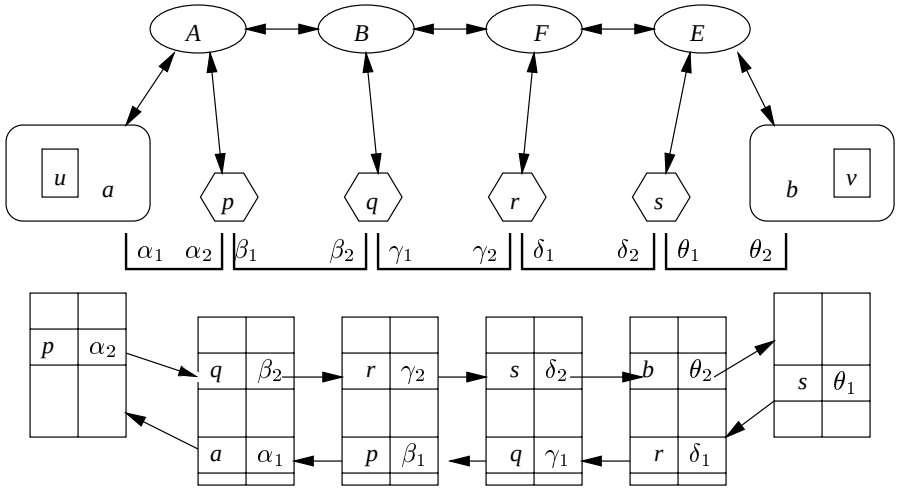


Fig. 2: Signalling and transport

```
4   extern char inbuf[];
5
6   void serve (int fd) {
7           int inlen = read (fd, inbuf, sizeof (inbuf));
8           (void) transform (inbuf, inlen, outbuf, outlen);
9           write (fd, outbuf, outlen);
10          close (fd);
11          }
12  ...
13  // in main:
14          int cd = context (STREAM_SERVER, O_RDWR, 0);
15          if (cd < 0 || cbind (cd, argv[0], 0) < 0) {
16                  perror (argv[0]); return cd;
17                  }
18          while ((int fd = copen (cd, 0, 0,))) {  // =accept()
19                  if (! fork ()) serve (fd);
20                  }
21
22  // ------------------------------------------------------------------
23  // Client using the stateless server
24  // ------------------------------------------------------------------
25  ...
26          int fd = cget (argv[0], 0);             // =connect()
27          if (fd < 0) {
28                  perror (argv[0]); return fd;
29                  }
30          write (fd, buff, strlen (buff));
31          int len = read (fd, buff, strlen (buff));
32  ...
33
34  // ------------------------------------------------------------------
35  // A multipoint application:
36  // ------------------------------------------------------------------
```

```
37          extern int mypeer (const char* hername);
38          int cd = cget (argv[0], 0);             // ~connect()
39
40          if (cd < 0) {
41                  perror (argv[0]); return cd;
42                  }
43
44          int fd = copen (cd, arg1, omode1);      // ~connect()
45          if (fd < 0) {
46                  perror ("failed connection"); return fd;
47                  }
48
49          for (;;) {
50                  int from, len;
51                  len = readx (fd, buff, strlen (buff), &from);
52                  writex (fd, buff, strlen (buff), mypeer ("signother"));
53                  }
54  ...
55  // ------------------------------------------------------------------
56  // A comparable MPI program (from the MPI specification)
57  //      note:   1. would be invoked with rank==virtual node# as argument
58  //              2. communicators are hard-coded numbers, not URLs
59  // ------------------------------------------------------------------
60  ...
61          char message [20]; int myrank;                  // rank == peer#
62          MPI_Comm mycomm = MPI_COMM_WORLD;               // ==default cd
63
64          MPI_Status status;
65          MPI_Init (&argc, &argv);
66          MPI_Comm_rank (mycomm, &myrank);
67          if (myrank == 0) {
68                  strcpy (message, "hello, world");
69                  MPI_Send (message, strlen (message),
70                                  MPI_CHAR, 1, 99, mycomm);
71                  // 1: the destination virtual address (peer#)
72                  // 99: a tag, to allow inband multiplexing
73                  }
74          else {
75                  MPI_Recv (message, 20, MPI_CHAR,
76                                  0, 99, mycomm, &status);
77                  // 0: the source virtual address (peer#)
78                  printf ("received: %s\n", message);
79                  }
80          MPI_Finalize ();
81  ...
82  // ------------------------------------------------------------------
```