# Moving beyond TCP/IP

*Fred Goldstein and John Day for the Pouzin Society 4/2010*

The triumph of the TCP/IP protocol suite in today's market is nearly complete. A monoculture of networking has emerged, based on Internet protocols originally developed in the 1970s and popularized in the 1980s. With this level of acceptance and a near-universal use of IP for purposes well beyond the original designers' intent, one can imagine that TCP/IP really represents the final stage in networking, from which all future solutions must slowly evolve.

This belief, however popular, is not necessarily correct. TCP/IP was a very important stage of network evolution. It was designed for the ARPANET, which the Department of Defense built as a resource-sharing network to reduce the cost of its research. It showed promise and conclusively demonstrated alternative forms of networking that had not previously been shown at such a scale. But it was not set up to primarily *do* research *on* network technology per se. There was no such network; research on networking was incidental to its mission.

It was, if anything, too successful. Because the ARPANET and TCP/IP worked reasonably well, was government-funded, and had no strong competition, it went into production too early. Its technology was adopted by the later Internet, as well as by many private networks, and it has been simply accepted as a given.

The Internet itself has been a huge popular success, in large part because of its low-priced business model. It hasn't been as successful for its providers; the ISP industry has never been very profitable, and the early growth of the Internet was largely subsidized by the huge infusion of overvalued equity during the 1997-2000 stock boom. IP has absorbed the glow from the Internet's halo. It is hard to distinguish between the Internet and its protocols. But they are not the same thing. For the Internet to prosper in the long term, it needs to move beyond TCP/IP. The experiment was a success. Now is the time to analyze its lessons and start moving ahead.

Unfortunately, the Internet Engineering Task Force has not been learning these lessons. It has been acting as a promotional body for TCP/IP itself. It confused commercial success with technical excellence. It has tragically decided that the evolution of IP is to IP Version 6. This decision, made in the early 1990s before the Internet was even in widespread commercial use, has distracted the networking community. A different direction is needed.

## *TCP/IP was designed for a limited set of tasks*

When the ARPANET began in 1969, it was designed to demonstrate the then-radical notion of packet switching. The mere notion that a network could handle small blocks of data, rather than create constant flows like a phone call, needed to be demonstrated. The original ARPANET protocol was called NCP. It was, in today's terms, *connection-oriented*: Before data could be passed between two points, a connection had to be set up. What differed from the connection-oriented telephone network was that the actual network capacity was only consumed on demand, not merely by having a connection present. But NCP was also a very rigid protocol, designed to ensure reliability of transmission on a hop by hop basis. This may have been a good idea given the networks of its day. The ARPANET backbone ran at 50 kilobits per second.. This was very fast at the time!

The NCP ARPANET is not today's Internet. It was closer to what the public packet-switched networks that were developed in the 1970s called X.25. These networks were optimized for terminal-to-host applications. It was a dead-end technology that dominated European markets in the 1980s.

But it was a French researcher, Louis Pouzin, who saw the early ARPANET and had a different idea about how to perform packet switching. He postulated that the switches in the middle of the network didn't have to keep track of connections; they just had to pass packets as they arrived. Error correction and flow control could be handled at the edges of the network. He designed and built the first *connectionless* network, CYCLADES, in 1972. He also noted in a 1974 article that it was possible for packets to be passed between separate networks, each with its own administration and policies. This idea was picked up later that year by ARPANET researchers, along with connectionless networking. We now know it as the Internet, the network of networks.

ARPANET researchers created a new set of protocols, with TCP for end-to-end error and flow control and IP for the connectionless middle. They did not receive universal acceptance even then. When submitted to the International Federation for Information Processing (IFIP) for international standardization, alternatives closer to Pouzin's work were found more suitable. The latter were submitted to ISO for its pending OSI program. But the Department of Defense continued with TCP/IP.

It was phased in and, by 1983's "flag day", completely replaced NCP. And around that time, Berkeley released a free, open source implementation of TCP/IP. It included the key application protocols of its day, too: FTP for file transfer, SMTP for email, and TELNET, which was largely used for remote terminal login. While designed for Berkeley Unix (BSD), it was adaptable to other systems too, and helped make TCP/IP popular. It worked, and while the original code was rather sloppy, the price was right.

TCP/IP's strength, as with its contemporaneous alternatives, was in dealing with bursty data traffic. It scaled reasonably well, thanks to Moore's Law, and with some rather critical adaptations made in the mid-1980s, adapted to high-speed local area networks. It easily handled new applications, including file and print services and the World Wide Web.

But it was not designed to replace every other network. Packet switching was designed to complement, not replace, the telephone network. IP was not optimized to support streaming media, such as voice, audio broadcasting, and video; it was designed to *not* be the telephone network. Packet voice was tested over the ARPANET in the 1970s, so the idea is not new. However, streaming over IP's "best effort" model of capacity allocation doesn't mix well with data traffic. This was recognized by the late 1980s and streaming packet protocols, with defined flows, were designed; the most successful was ST-II (see RFC 1190). But these didn't catch on. Instead, the rapid growth of capacity in the Internet backbone allowed streams to just hitch a ride, working just well enough to catch on. It was a case where good enough was the enemy of the best. The IP juggernaut was unstoppable.

## *IP has a number of weaknesses today*

TCP/IP was a useful laboratory experiment, and it helped the Internet to get started. But that doesn't make it perfect. The TCP/IP ARPANET itself had weaknesses that are surprising in a network financed by national security dollars. It lacked any kind of security mechanisms, depending entirely on the security of the computers connected to it. (We see today how well that

worked out!) And it lacked support for redundant connections (multihoming), even though the need had been identified in the early 1970s. This wasn't rocket science; by the 1980s, TCP/IP was the most open protocol stack but not the most powerful.

And the Internet of today isn't the ARPANET of the 1970s or for that matter the Internet of the early 1990s. It scaled up pretty well, but not infinitely. So it's time to size up the lessons learned and start on the next generation of networking. TCP/IP was a research project; let's exploit the results of that research. There are many issues that IP doesn't handle well and which could be addressed if a new protocol were developed from a clean slate, rather than as an extension of IP.

## IP doesn't handle addressing or multi-homing well at all

The original ARPANET was designed for the relatively small production loads of its day. It was secondarily a tool for developing packet switching technology. The ARPANET was also designed around attaching each computer (a *host*) to only one IMP (Inter-Machine Processor, as its early packet switches were called). The NCP address was the IMP's ID number plus the port number going to the host – just like a primitive phone switch, with its exchange code and line number, and the early datacomm networks, which focused on terminal to host connectivity. This turned out to be a problem as soon as an ARPANET host asked for a redundant connection, and as a result ended up with two addresses. When IP came along a few years later, it inexplicably kept this convention, addressing the interface, not the host. But what happens when a host needs a reliable connection, and thus needs redundancy? Because IP addresses are assigned to each interface, a multihomed host looks like two different hosts. This makes multihoming almost useless, as traffic bound for the failed interface doesn't know about the alternate path. It's serious mistake, and the solution is not hard. This is no different from having logical addresses and physical addresses in computer operating systems. Other early networks such as CYCLADES (in 1972), DECnet (around 1979), XNS and OSI got it right. In fact, essentially all of the early packet-switched networks except the ARPANET and then the Internet got it right. If the multihomed host itself had one address, routing algorithms could easily calculate alternative paths to it. Oddly, the convention was maintained by the IETF in IP Version 6.

The problem is magnified when dealing with multihomed *networks*. The mythology about IP is that it was designed by ARPA to survive nuclear war. The reality is that it has trouble figuring out where to send packets at any given time. This comes about in part because its addressing scheme is next to nonexistent! Multihoming is not its only problem.

The first part of an IP address is the *network* part, which indicates the network that the interface is attached to. These numbers are really just variable-length numeric names for networks. Because of the way they have been historically assigned, two adjacent network numbers can be in different parts of the world. So every backbone router needs to keep track of every network. Hundreds of thousands of them. All the time.

In other words, it's not an address at all. It's like giving every house in a large city its own number, in order of construction. Not 100 Main St. or 301 17th St., but 3126, while 3127 is halfway across town and your next-door neighbor is 188. The actual location, which an address should convey, is computed in each router, in a very data-intensive process. The rest of the address identifies the interface within that network. It identifies the *subnet*, which is used for

routing to a group of nodes within the network (akin to a street), and finally the specific interface (akin to a house).

Since 1992, an interim fix has been to assign most companies provider-dependent addresses: If a company attaches its private network to one ISP, then it gets its IP addresses from its ISP. This connection is invisible to the outside world since only the ISP's network identifier is flooded. But if it wants to connect to two ISPs, it needs its own *provider-independent* (PI) address block. It's like being its own city in the postal address system. It now becomes part of the backbone; its block is one of hundreds of thousands that need tracking.

Backbone routers then exchange information amongst each other about all of the routes and subnet blocks that they serve. This doesn't scale well at all. The more networks on the Internet, and the more multihomed PI address blocks, the more routes there are, and the number of possible paths thus rises *faster* than linearly. And as the Internet grows more important, more and more companies will want multihoming and PI address blocks. This has been good for the router makers who need to sell newer, more powerful routers just so that network operators can keep up. It's bad for everyone else. But even the router vendors are having trouble keeping up. Oops.

It has been suggested that as many as ten million businesses may eventually want multihoming. The core IP network simply cannot sustain that many routes. Now let's think about another up-and-coming application, the electric industry's "SmartGrid". This potentially involves having every electric meter, at every home and business, be on-net, multihomed. Each major utility thus has more than ten million of its own potential multihoming devices. Worldwide, this can add up into the billions.

The IETF's proposed solution to the multihoming problem is called LISP, for Locator/Identifier Separation Protocol. This is already running into scaling problems, and even when it works, it has a failover time on the order of thirty seconds. This is bad enough for web browsing but if it were used for the electrical grid, it's enough time to lose the east coast.

How does IP Version 6 handle this? By having a larger address space, it permits more address blocks to exist. That's all. It doesn't change the architecture; it just speeds up the fuel pump feeding the fire. Perhaps it's moving in the wrong direction. Maybe that's why the authorities that grant blocks of IP addresses are being so parsimonious with IPv6 blocks. And it's not as if they didn't know better. IPv6 was designed after an attempt to correct the problem, sometimes referred to as IPv7 (also called TUBA), was designed, deployed, and tentatively agreed upon as the successor to IPv4. IPv6 was designed to maintain the status quo.

## TCP and IP were split the wrong way

TCP and IP started out as just one experimental protocol, called TCP. It was split into two in 1978, with IP separated, in recognition of the fact that routers only need to look at the IP header, not the TCP header. The first widely implemented form of this was TCP version 4, which is why the split-off IP uses that version number. So they looked like separate layers to the ARPANET community at the time. The concept of layers as black boxes came from Dijkstra's work on operating systems, which was well known by 1970.

IP's key function is relaying packets to the next hop. TCP has two key functions. One, which it shares with UDP, is identifying the connection: The port ID identifies the instance of communication between two hosts. TCP also provides error and flow control. The problem is that *mechanism* and *policy* are not separate. TCP implements a single policy with its mechanism; UDP implements a second policy with its mechanism.

TCP provides the feedback from receiver to the sender that allows the sender to retransmit lost packets, effecting end-to-end reliability. But TCP's header sends these feedback messages in the same packets that carry data forward. Processing would be considerably simplified if control and feedback headers were kept on separate packets from those that carry payload. On a high-speed network, the payload needs to be processed rapidly. Control can run at a more leisurely pace, tied to the round-trip time of the connection. This "piggybacking" was an optimization for the very short packets that dominated the ARPAnet in the 1970s, when the killer application was logging in to remote computers from a dumb terminal that needed character-at-a-time echo, and thus had largely symmetric flows. But it makes processing more complex today, with few if any benefits. Elements that need to be associated with the data (relaying forward) have few interactions with those that don't, such as flow control messages and the acknowledgments that, when not received in a timely manner, prompt retransmission.

Another problem is that it numbers and acknowledges bytes, not packets, the way almost all other protocols do. This is an artifact of the early TCP, before IP was split off; it facilitated fragmentation within the network. But it makes it harder for TCP to scale to higher speeds, especially over imperfect or long-delay links. Associated with this is the general inability of TCP/IP to fragment packets reliably, with the layers split as they now are. Different physical media and data links have different maximum packet sizes. TCP at best copes with this by using a discovery technique that sees what packets do and don't get through. But this doesn't get along too well with IP networks that are allowed to change the path.

## IP lacks an addressing architecture

IP itself, and the TCP/IP stack in general, lack an actual addressing architecture. Most telling is that the IP address is not even an internet address at all! Because it identifies a point of attachment to a host, it is basically a data link address. The host itself is not addressed. That would be the proper role of an "Internetwork" layer. This was identified in the mid-1980s as part of the research for the OSI program. The network layer ("layer 3") was identified as having three separate "roles". The bottom one (subnetwork-dependent) dealt with the underlying network (such as a LAN or a public packet-switched network), the top one (subnetwork-independent, or "internetwork") dealt with the overall network, and a convergence function mapped the two together.

Prior to 1983, the ARPANET ran NCP, and when TCP/IP was first introduced, it ran atop NCP. In that era, IP really was an Internetwork layer, and NCP was the subnetwork. But when NCP was shut down, IP took over the network role. It was no longer a true Internetwork Protocol; it became just a network protocol. No wonder it has problems with multihoming, mobility, and other address-related matters.

## NAT is your friend

This brings us to another major controversy in the TCP/IP world, Network Address Translation (NAT). This serves two major functions. One is to conserve IP addresses, which is of course important in a world running out of IPv4 addresses, This also applies to home networks that have multiple computers sharing a single ISP connection that provides one IP address at a time. The other function is security: Devices with private (NAT) addresses can't be directly addressed from the public Internet. The NAT gateway acts as a firewall and thus protects against some kinds of attack. For this reason, many corporate networks use NAT even though they have plenty of their own PI address space. Only secure public-facing servers go outside of the firewall.

NAT is often seen as a layer violation because the NAT device has to modify the TCP *and* IP layers together, changing port numbers (in TCP or UDP) as well as IP addresses. This is not a problem with NAT per se. IP address + port number is the connection identifier. So address translation naturally deals with them together. NATs only break broken architectures. Consider this a lesson learned from the great TCP/IP experiment.

There is, of course, one clear layer violation that NAT has to deal with, but that's not NAT's fault either. Some application protocols put an IP address inside the application layer header. These have to be modified by NAT, so a NAT has to understand the syntax of all of these layer-violating applications that it supports. The original application that did this was FTP, going back to the very early ARPANET days. FTP did this because – remember, this was a lab hack, not meant for production – BBN, who built the original packet switches, had line printers attached to user ports on its Terminal Interface Processor, an early terminal server. The TIP didn't have enough memory for a name table, so the protocol port number in the TCP header literally identified to the physical port on the TIP. (That's why they're called port numbers.) This problem was fixed years before IBM built its first "PC". But other protocol designers assumed that FTP's designers must have known something they didn't, so they blindly copied it. This is rather like passing physical memory addresses in a Java program.

SIP, the popular VoIP control protocol, also allows IP addresses in the header in lieu of names. This is done because telephone sets and other devices may not even have name entries in the DNS, especially if they're behind a NAT. This points to a second problem with TCP/IP. The real name of an application is not the text form that humans type; it's really identified by an IP address and its well-known-port number. When an application uses a host name or other text form such as a URI, it is the application that resolves it by querying DNS. This should instead be done by the network layer machinery in the host. That would close off the excuse for putting IP addresses in the application layer, and provide a uniform mechanism for application-entities to find each other. Again, this was not recognized by the ARPANET developers in 1975, when there were only a small number of hosts on the net and only a few applications, but it's a case where the TCP/IP architecture has not scaled well.

In fact, even host names should not be used in the application. Applications, not hosts should be what's named! The host simply, er, hosts it! Many applications nowadays run on more than one host; this requires a royal kludge in the TCP/IP architecture.

## IP is overburdened by local peering

The original “Internet” concept was one of large-scale networks with limited interconnection between each other. The term dates back all the way to CYCLADES. It came into widespread use when the Department of Defense separated its internal MILNET from the more research-oriented ARPANET. The two IP networks remained linked, but under separate administration. The pre-public ARPANET and later the Internet grew by adding organizations such as universities and corporations, each of which managed their own network and which typically had one or two links to the rest of the network outside of the company. Most users had more of a community of interest within their own organizations than with the rest of the world; public Internet access was thus a feature of their network, not the goal itself.

Today’s Internet is used very differently. It functions, like the telephone network, as a consumer and general business service. Corporations often use encrypted links across the public Internet to link their own sites, rather than use private links to access the Internet centrally.

But because of the way IP works, interconnection between networks remains largely centralized as it was in the 1980s. With some narrow exceptions, each link between networks must be reported to every router worldwide. So if Comcast in Massachusetts were to create a direct link – that is, peer – with Verizon in Massachusetts, this link and its current status would be reported via the backbone’s “flooding” system to routers in New York, Texas, and Uzbekistan. So network designers have to strike a balance between creating too many links, and thus overburdening the routers, and having too few links, and thus having to send local traffic on a very roundabout route.

The telephone network doesn’t work like this, of course. Within the United States, local calls are almost always exchanged within each LATA. The presence of an additional link between two carriers is invisible, and these links number at least in the tens of thousands. The primary path through an access tandem is reported through a data base distributed to subscribers monthly; other paths are private. And that is mainly of interest to other carriers within the LATA. So local phone calls go through one or two hops from end to end, while long-distance calls usually need three or four. To be sure, the telephone network dates back to the days when “long distance” was very expensive. But most calls are still local, and it’s still pretty efficient for calls of all distances.

Contrast this, using the traceroute program on almost every computer, to the way ISPs interconnect. A packet from RCN in Boston to Verizon or Comcast in Boston goes through New York. Some local packets hairpin through New Jersey or Virginia. A link of *less than* ten hops is a rarity, unless it’s to one’s own ISP or a content distribution network collocated on that network. This inefficiency adds to the cost of service and degrades its performance, but it’s inherent in the way IP works. It simply wasn’t designed to replace local public networks. Keep this issue in mind when envisioning a possible transition to an “all-IP” telephone network. One workaround is to have private peering relationships that do not get reported to the backbone. These handle some high-volume routes, but IP’s architectural limitations makes local peering the exception, not the rule.

## IP is poorly suited for streaming

IP was designed to deliver packets on a “best efforts” basis, meaning that it’s okay to throw packets away. That’s not a bad thing. But it works best when the payload can use

retransmission. Streaming means that there's no time to retransmit, so the stream must have low loss. The best that IP can do for streaming is to assign priorities, with high priority given to streams, such as telephone calls, that don't tolerate loss.

Standard TCP data slows down using the slow-start algorithm; streams do not. So if there are too many streams on a link, ordinary data can be crowded out. We already have a telephone network and cable TV; these applications could potentially break the unique capabilities that only the Internet can provide. The telephone industry is evolving towards the use of IP, but it is a tricky proposition. The telephone streams often have to be separated at a lower layer, put into separate flows using MPLS, Carrier Ethernet, or some other technique, even TDM. These offer the lossless assurance of bandwidth that IP, being connectionless, lacks. Attempts to handle streaming within IP are preposterously complex and unproven. (See, for instance, IMS, the IP Multimedia Subsystem. It's the nuclear fusion of IP: It's always a few years from being ready.)

When these lower layers are doing the heavy lifting, what's IP's role? Mostly it's just a way to multiplex the traffic, and to take up space, while getting the alleged coolness factor. and possible regulatory advantage, of IP. In one prescient Lucent Technologies flyer from the boom years, a short-lived VoIP add-on to its 5ESS telephone switch was said to have, among its key benefits, "Wall Street image" and "access to capital". The truth is that while voice and video *can* migrate to IP, it's a marriage made in hell, as it is difficult for both to truly get along without a lot of waste. While the FCC has been asking about a perceived migration towards an all-IP telephone network, it has clearly not yet distinguished between the Internet as a regulatory and business model, whose favorable regulatory treatment has driven the migration, and the IP protocol itself.

## A path forward

So if IP is so imperfect, can anything be done about it? Of course… but it's not going to be handled by incremental upgrades or missteps like IPv6. Instead, what John Day has done in his *Patterns in Network Architecture: A Return to Fundamentals* is start afresh, taking into account lessons learned in the 35 years of TCP/IP's existence, as well as the lessons of OSI's failure and the lessons of other network technologies of the past few decades. He has made some key observations that point to a new direction.

Day has been on the net since 1970, and participated in early work on TCP, TELNET, and FTP, as well as the IFIP working group referenced earlier. He was, for some years, the rapporteur of the OSI Reference Model. That is, he chaired the committee that was charged with developing the famous 7-layer model which had itself been invented in the late 1970s. Early on, the OSI upper layer group recognized that the model itself was fundamentally wrong! Layers 5, 6 and 7 were really one layer, and did not work as separate ones. This was known by 1983 and the protocols were refined to allow them to be handled together. But not all implementers or textbook authors understood this, and attempts to build OSI-compliant applications atop separate layer 5 and 6 protocol machines met with so much difficulty that the whole OSI program collapsed. And indeed one of TCP/IP's strengths was that it lacked these superfluous layers. The IP community of the day was if anything practical, apparently more so than it is today.

But fixed protocol stacks themselves turn out to be the problem! The *pattern* that Day noted is that protocol functions tend to alternate as one goes up the stack. This alternation reflects a repeating unit consistent with interprocess communication. The *error and flow control* protocol

breaks up into two functions, *data transfer*, which sends data forward, and *data transfer control*, which provides feedback from the receiver. These functions both happened in both X.25's layer 2 and 3 protocols, and it also describes the relationship between IP (data transfer) and TCP (data transfer control). Applications themselves have a common pattern too.

Lower layers thus tend to do the same things as upper layers, but on a more local scale, with more flows aggregated into them. At the application layer, a single instance frequently communicates between two computers. At the other extreme, a large network backbone link may carry thousands of processes, headed between many different computers, part of the way.

## Recursive layers

This leads to the first principle of our proposed new network architecture: Layers are *recursive*. The same protocol can be used repeatedly in a protocol stack, encapsulating each layer in another instance of itself. There is thus no need for purpose-built protocols for each layer. There is also not a fixed number of layers in the stack. The number of layers between the application and the physical medium is variable; at any given point, there are simply as many as needed, no more, no less. But any given implementation only needs to deal with itself, the layer above it, and the layer below it. The actual depth of the stack is essentially invisible.

Layers are not the same thing as protocols; more than one protocol can make up a layer. Because the same group of protocols is used repeatedly, the implementation is simpler than the TCP/IP stack. There's no need for separate protocols for "layer 2", "layer 3", etc. Because the layers recurse, and can scale to form a large internet, the protocol suite that supports the concept from *Patterns in Network Architecture* (PNA) is called the *Recursive Internetwork Architecture* (RINA).

The protocols that make up the basic RINA layer include the Data Transfer Protocol (DTP) which relays the payload in the forward direction. Its contents includes addressing information and protection information (a checksum and a time-to-live counter which detects routing loops). The Data Transfer Control Protocol (DTCP) performs the error and flow control functions, sending feedback from destination to source. Note that DTP has the payload, which is visible outside of the layer, while DTCP operates entirely within the black box. A layer also has a management protocol.

Another observation that Day made is that networking is just interprocess communications (IPC), a standard function on modern computer operating systems. IPC within a single computer is quite simple; it could just take the form of memory shared between two processes. IPC between computers requires additional mechanisms to deal with issues such as reliability of communications and synchronization. That's where network protocols come in.

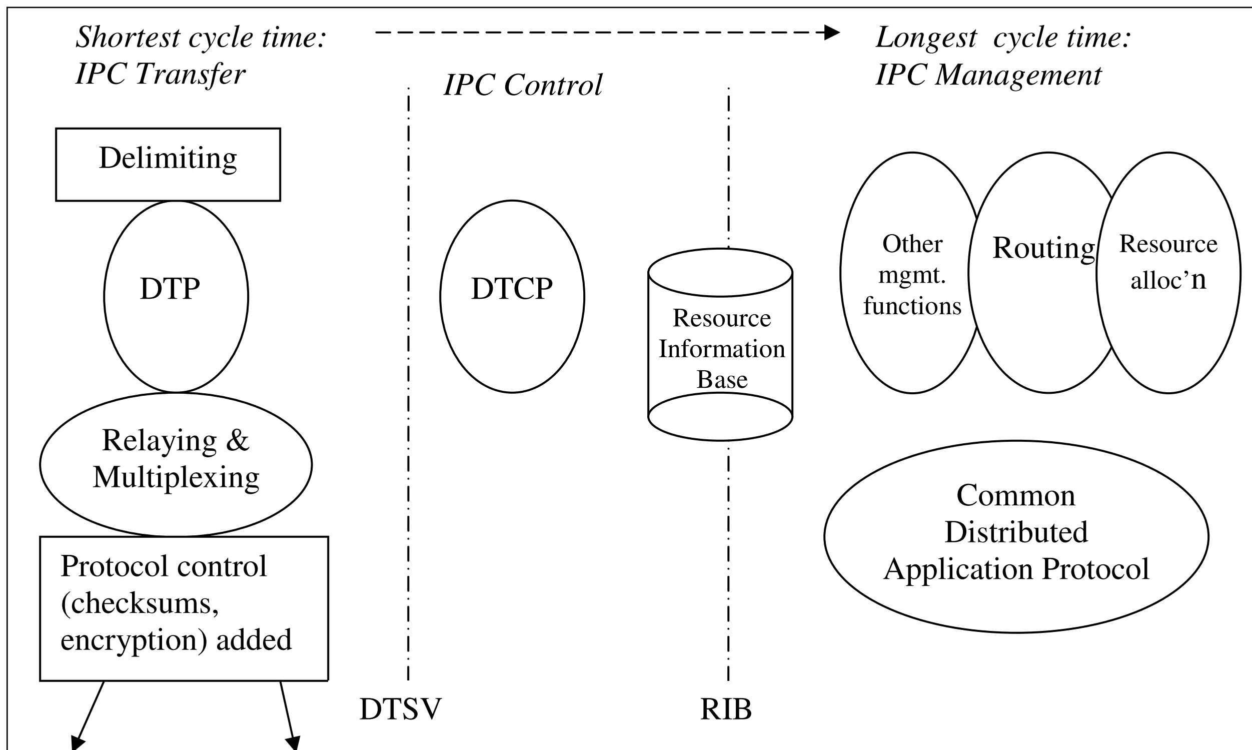


*Figure 1Functions are executed within a Distributed IPC Facility over different time scales.*

Hence the combination of functions that make up a layer is called a Distributed IPC Facility, or *DIF*. An instance of DTP and an instance of DTCP fall within a single DIF, a single layer. DTP functions operate over the shortest time frame. DTCP functions operate over a slightly longer time frame; the two share information via the Data Transfer State Vector (DTSV). The error and flow control functions of DTCP might only be performed at the edges, as with TCP over IP, but there's no formal layer boundary between them.

A DIF also includes a number of management functions, which among other things include setting up routing within the DIF. These functions, which are somewhat less time-critical, are stored in a Resource Information Base (RIB).

The DIF is the basic mechanism of RINA. It is a black box that operates among multiple systems; the IPC process thus runs on every system that belongs to the DIF. A DIF enforces strict layer boundaries: What happens inside the DIF is not visible outside of the DIF. What is visible at the top of a DIF is the *service* requested of the DIF; what is visible at the bottom is the service requested *by* the DIF of the DIF beneath it, if it isn't the bottom one.

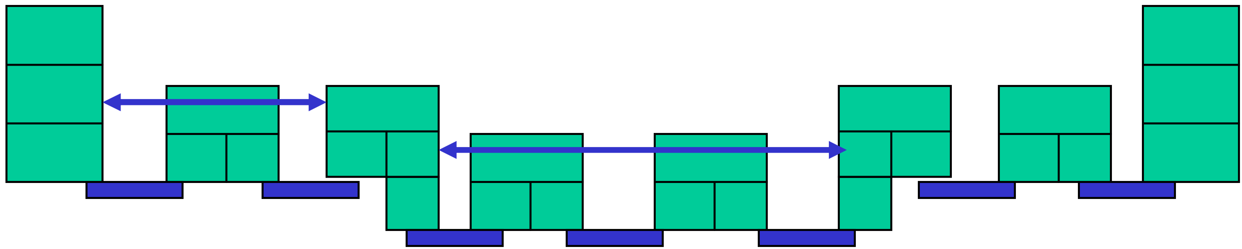

*Figure 2 Distributed IPC Facilities may be stacked as required. The number of layers in the stack may vary from hop to hop*

An application process is at the top of the stack. The DIF beneath it is aware of it, and identifies it by name. A DIF can span many systems; it may rely on the services of lower-layer DIFs to link the members of the DIF. And these DIFs may in turn rely on DIFs beneath them, transparent to the application processes that use them.

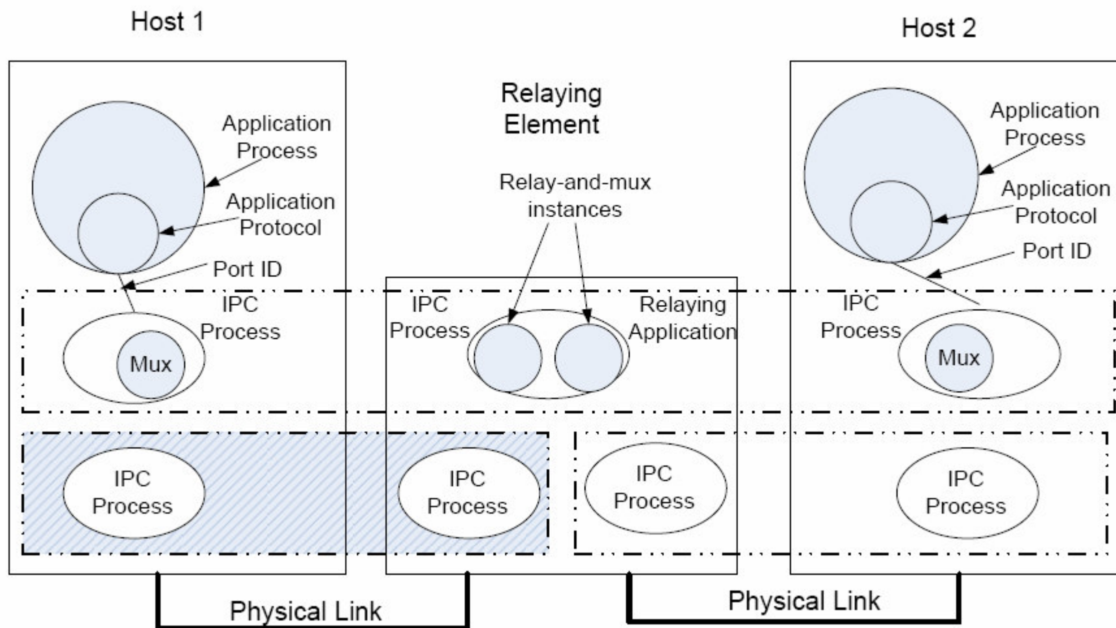


*Fig. 3. A router is a DIF which performs relaying between DIFs as its application.*

One of the fundamental DIF services is the encryption of its payload. It's impossible for a relay to "peek inside the envelope" of an encrypted DIF. Hence there is no need for a separate RINA equivalent of IPsec or SSL, because encryption is available *everywhere*. Deep packet inspection is thus impossible. Network *content* is thus inherently neutral, though not in the "one size fits all" sense desired by many IP neutrality advocates.

The basis of DTCP is the delta-T protocol, invented around 1978 by IBM's Dick Watson. He proved that the necessary and sufficient conditions for reliable transfer is to bound three timers. Delta-T is an example of how this should work. It does not require a connection setup (such as TCP's SYN handshake) or tear-down (like TCP's FIN handshake) for integrity purposes. In fact,

TCP uses the same three timers! So RINA lacks unnecessary overhead. Watson showed that synchronization and port allocation are distinct functions. In fact, maintaining the distinction also improves the security of connection setup.

RINA also only requires a single Application Protocol, the Common Distributed Application Protocol. Just as there are patterns visible in the other layers, all applications can be shown to need only six fundamental operations that can be performed remotely: Create/delete, read/write, and start/stop What changes from application to application is in the objects being manipulated, not the protocol. So the protocol remains stable, while new applications are accommodated easily by changing the object models. The common application protocol is used within the DIF for layer management functions. These include enrollment (initializing the DIF) and security management, port allocation, access control, QoS monitoring, flow management, and routing.

## Designed for multiple functions

TCP/IP was designed for the data networking functions of the day, and was *not* intended to replace the telephone network or for that matter the cable TV network. VoIP and IPTV are essentially done by brute force. RINA starts by asking the question, can a protocol be effective for multiple functions with very different requirements? The answer, of course, is yes. It's not hard to effectively serve different applications if that's the goal from the beginning of the design.

RINA does this by having many plug-ins that control how the DTP and DTCP work. One of the key differentiators of RINA vs. TCP is its support of Quality of Service (QoS) options. These include parameters such as allowable packet loss rate, error rate, delay, and jitter (delay variance),. A DIF *can* operate in a "best effort" mode, like IP. In a recursive stack, only the top DIF might need to do the end-to-end error correction. That's how TCP/IP works and it's fine for many applications.

In RINA, *policy* and *mechanism* are separated. The mechanism of a protocol may be *tightly bound*, such as the headers of a data transfer packet, or *loosely bound*, as in some control and acknowledgment messages. Different applications may need different policies. Flexibility comes from being able to use common mechanisms in conjunction with different policies, rather than needing separate protocols. The *syntax* of protocols dictates neither policy nor mechanism. Protocols can thus be expressed in an *abstract* syntax, which is then converted to the *concrete* syntax (the bits on the wire). This is more flexible than the rigid bit-field syntax of the TCP/IP protocols, and enables the same basic mechanisms to be used in very different ways. For example, the question of address field size is simply not something to argue over: The length of an address is part of the concrete syntax, and it just needs to be appropriate for the given DIF at the given time. It is not a constant. (Recall that the only actual address of a DIF is a name; the "address" field is a short alias that facilitates routing within the DIF.)

Once policy is separated from mechanism in data transfer protocols, the only major differences are syntactic, and these are limited to the lengths of the address (alias), port-id field, and sequence number related fields. They have little effect on how the information in the fields is used. The port-id is at most an index into a table; the length of the sequence-number-related fields only affects the modulus of the arithmetic. Otherwise, the procedures using them are the same. These differences are hardly warrant writing entirely different protocols for each case. So the syntax for all data transfer protocols can be reduced to a small number of cases, with common procedures and plug-in policies.

Protocol historians studying the X.25 era might remember an earlier construct that foreshadowed this. An HDLC connection could be initiated with the "SABM" command, providing a 3-bit sequence number field, or with the "SABME" command, with a whopping huge 7-bit sequence number. (The smaller field saved a whole byte in each packet, making it very popular, and it could usually keep up with a teletype terminal.) Different carriers' networks had different policies about which form to support.

Various QoS options have existed in other protocols, such as ATM (asynchronous transfer mode). QoS has gotten a bad name in the IP community, but then it's impossible to deliver anything but a best-efforts QoS on pure IP. This comes about because QoS requires metering of the data rate, which is not part of IP. One alternative in IP is to establish priorities, or "Class of Service" options; another is to create connections below the IP layer and map specified IP streams onto these connections. That's what MPLS does, for instance.

A DIF can be asked to provide a low-loss connection for a specified level of capacity, such as might be required for an audio or video stream. Or it can be asked for a best-effort (unspecified QoS) connection. RINA simply considers QoS to be a set of parameters plugged in to the DIF, and the specific mechanisms that it uses to provide it are an implementation detail, hidden inside the black box. If the DIF determines that it can't deliver the requested QoS, it rejects the request.

## Integrating connectionless and connection-oriented networking

One of the reasons that many TCP/IP backers are so passionate about their cause is that they fought hard battles over it. In the early 1980s, the OSI committees faced a serious conflict between those who supported connection-oriented networks, and those who supported the connectionless approach. OSI defined a "connection-oriented network service" (CONS) that was based closely upon X.25. It promised "reliable" transport, which meant that the network itself was involved in error and flow control, and solicited retransmissions when it detected a dropped packet. It also defined a "connectionless network service" (CLNS), not all that dissimilar from IP. (In fact, it was the OSI Connectionless Network Protocol (CLNP) that was proposed as the basis of the next generation of IP, to be called IP Version 7, which was already implemented in the major routers of the day. This was in the early 1990s. This decision faced serious opposition from the rank and file of the IETF, who saw OSI as the enemy, a competing standards program, and that led to the eventual development of the dramatically-inferior IP Version 6.)

Connectionless service is fine for many things; the TCP/IP Internet is an existence proof that it works, something that the early CONS backers would not have believed possible. But the OSI CONS gave connection-oriented networking a bad name. It tried to do too much inside the network. It wasn't until years later than "lightweight" connections took hold.

As it turns out, the distinction between the two is less than it seems. The usual argument for connectionless networking is simplicity. By not promising too much, the network is simpler, or "stupid". A protocol has been considered connection-oriented if it needed to set up a context for the flow of data. IP is considered connectionless because each packet is just routed as it comes along. But wait: How does a connectionless (IP) router know where to route packets to? We've already determined that an IP address doesn't actually tell you where it's going; it just names the destination network and node. So in fact connectionless routers require *more* context than connection-oriented ones, because they need to maintain, in effect, connections to *every other node on the Internet*! That's what routing tables are all about, and why they're getting bigger by the day.

In fact, connectionless networks do have an equivalent of connection establishment, but it happens well before packets are sent. Port numbers are allocated (some have been allocated for decades) and addresses are assigned to hosts. This is essentially the enrollment phase of a connection. Enrollment is required in RINA too, of course. A RINA user can enroll in as many DIFs as it needs to. A DIF is a collection of applications working together. Relaying packets (what a router does) is simply one application, albeit one that provides service to DIFs which operate at higher layers. If a computer needs to reach an application that is not on a DIF that it belongs to, it can query a database on a DIF that it does belong to; that database may know what DIF to join to reach that application, and may give routing information about how to get there.

So the connection-oriented vs. connectionless wars turn out to have been much ado about nothing. What RINA provides is a set of connection options that are fundamentally lightweight. The DIF is in some ways connection-oriented: Before a connection is established across a DIF, at any layer, the peer entities authenticate the application. (All connections, at all layers, are applications of a DIF.) Requiring authentication provides security: If the recipient of a connection request does not approve, then the connection is rejected. (Take that, spammers. Sayonara, denial-of-service attackers.)

But unlike TCP, the delta-T-based DTCP doesn't require explicit connection establishment or tear-down messages. So it has less connection overhead than TCP/IP. Since the same DIF protocols are used recursively, and end-to-end error control usually only has to happen once, many of the functions of EFCP simply aren't requested except at the ends of a connection. The DIFs that provide intermediate relaying might only need to provide best-efforts QoS. But if the application is a stream, end-to-end retransmission is not possible, so in that case all of the DIFs in the stack are likely to be asked to support a streaming QoS.

## Names are global, addresses local

IP, as noted before, actually uses IP addresses as part of an application-entity name. The domain name system is merely a human-friendly index that returns, within the application layer, a numeric IP address. One of the newest major protocols in the TCP/IP suite, HTTP (invented in 1989), tries to improve upon this; the actual URL string, a name, is transmitted to the web server, even as the IP layer uses the numeric address returned by DNS. This lets a single web server, with a single IP address, be shared by many different names.

The 32-bit IP version 4 address was more than adequate for the ARPANET. It would have been sufficient forever, had it not been used for the worldwide public Internet, with billions of personal computers as clients. The IETF's approach, IP version 6, "fixes" the shortage of addresses by putting a 128-bit source and destination address in every packet. This simply perpetuates IP's architectural flaws. The correction question is why? Why should there be a single worldwide numeric address space?

The obvious precedent is the telephone network, which has a worldwide numbering plan. But telephone numbers nowadays are really names, just names designed to be entered on a numeric keypad. Most calls require a quick database lookup before they are completed. The actual address (such as the LRN) is hidden within the network. In RINA, connections are requested of a DIF by name, and only by name. A name is a variable-length string, typically fetched by the application; it need not even be visible to the end user. There is no equivalent of the IP address. Every service directly attached to a DIF is known to that DIF, by name. Everything else is reached by relaying to another DIF. Names are thus passed from DIF to DIF as part of the

connection establishment phase. A DIF names its own services points; its points of attachment are the names of the underlying DIFs. (Compare this to IP which names points of attachment in lieu of a proper internetwork address.)

Addresses of other forms may exist *within* a DIF. The scope of an address, then, is the DIF itself, and the address is really only an alias for a name. Since it's not visible to the end user, it is only useful if it serves some purpose inside the black box. Hence the address is likely to be *topological*: It conveys information about how to reach the destination. Two adjacent addresses are likely to be very near each other. An address itself could be computed based upon the graph of the network. This is most useful for large DIFs, of course, and it doesn't take the place of a routing algorithm; it merely simplifies the task.

The size of the address is a parameter of the DIF. A small DIF (like a LAN) may just use very short addresses, with each node knowing how to reach every other one. A point-to-point link is itself a DIF, one which doesn't need an address at all. At the other extreme, the entire World Wide Web, or all of the public servers on the Internet, could be cast as a DIF, and it might need relatively large addresses. But they would still be invisible to the application. An implementation of RINA would take care of addresses automatically; they would not be assigned via human intervention, or subject to an addressing authority.

## Search and you shall find

TCP/IP's Domain Name System was a rather rudimentary database application even when it was created in the mid-1980s. By today's standards it is hopelessly primitive. In a global Internet, how would RINA handle the task of finding a destination name? Unlike IP backbone routers, which all need to know the path to everyone else, RINA routers can query a database, which can return information about how reach the destination – for instance, what DIF it's on, and what DIFs connect to it. The database is modeled on a search engine. It need not return a single answer, since there may be more than one way to get to the destination. But by separating this part of the routing function from the relaying function, the router (a relaying DIF) can be dramatically simplified.

A RINA database need not contain the entire world's Internet; it only needs to list the members of its own DIF, though it may choose to also list paths to others. Since there are no global numeric addresses (like IP) to attack by brute force, the database can act as a first line of security: It need not return any answer at all to an unauthorized user (think "intranet"), or could indicate that the user's connection request is unlikely to pass authentication.

## Mobility is inherent

A benefit of this database-and-name-driven system is that a given application could have a "home" database search engine, and its actual current location could be updated in real time. Routing instructions are provided by search engine. This can be dynamic, more like cellular telephony's home location register than like the static DNS. If you are using a mobile computer, its application-names would be constant, and as the computer moved from network to network, it would re-enroll in the new networks (perhaps automatically), and continually update this information in its home search engine. There is thus no need for the "triangle" routing used in mobile IP. Even cellular telephony has a triangle of its own, as the call is routed through the dialed number's home switch before the home location register is queried. In RINA, routing is fully dynamic.

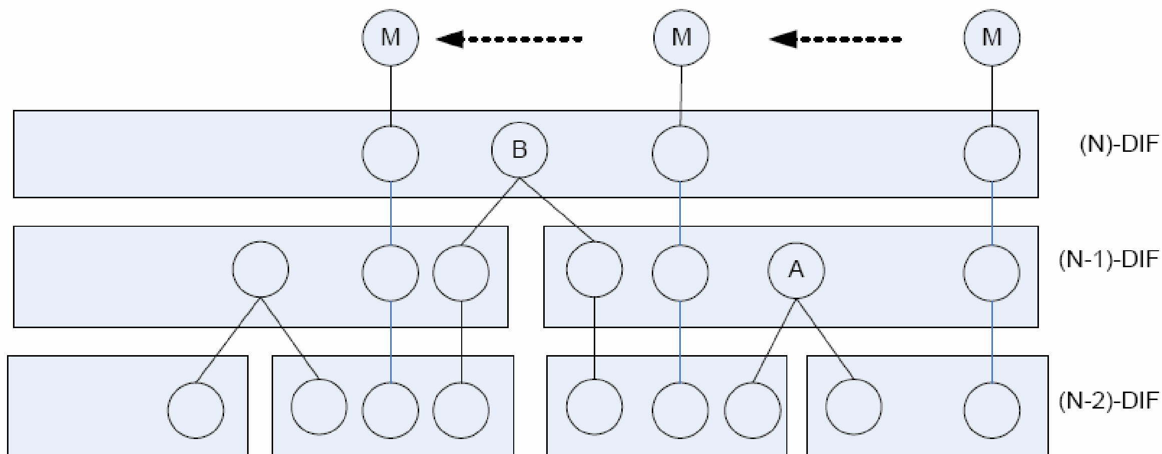


*Figure 4. As a mobile host moves, it joins new DIF and drops its participation in old ones.*

## Unicast, multicast, and anycast

IP addresses do not refer to applications or hosts; they refer to a point of attachment (actually, a data link address!) on a single host. It was a perfectly reasonable way to do an experimental network in the 1970s, but today, a DNS entry often points to a system that quickly redirects users to a member of a group of hosts which can collectively handle the actual load.

RINA names are more flexible. The most obvious form of reference, of course, is unicast: The name refers to an application resident in a single place. RINA does not name even attempt to name hosts per se. It names *applications*. An application can live in one place, which is addressed via unicast. But often an application resides on more than one host. For instance, "http://www.google.com/" does not actually live on a single host; it is redirected somewhere, invisible to the user. In so doing, it is simulating a second type of address, *anycast*. Any one of a *set* of destinations can serve the request. (Once a dialog is established between two hosts this way, it is likely to be completed via unicast.) The third case is *multicast*, in which the information is relayed to every member of a set of destinations. While multicast nominally exists on the Internet, it is rarely implemented; it was an afterthought, not easily handled.

Multicast and anycast turn out to be subsets of a single form, *whatevercast*. Both deal with a set of addresses and a rule. Anycast addresses one member while multicast returns all members that satisfy the rule. Just how a DIF implements whatevercast is an internal matter, of course, since it's a black box, but the general case is that a name refers to a set of destinations. Hence unicast can be seen as the special case, one where the set has a single member; the application is in one place. Thus applications need do nothing special to deal with multicast or anycast.

The value of efficient multicast is obvious in the case of streaming video. A video stream can be sent to a multicast set of destinations. This set can be dynamic: When a user wants to watch the stream, the set top box or computer is immediately enrolled in the multicast set. Only one copy of the stream is relayed *to* any given DIF, though that DIF may use its multicast capability to relay the stream to as many other DIFs as necessary to reach the members of the set. This is far more efficient than Internet TV using UDP over IP, in which every viewer is watching its own

stream from a server. It could even provide a practical way for cable TV to evolve to a common structure with the Internet itself.

## Easier adoption than IPv6

It's one thing to describe a protocol in the abstract and quite another to put it to good use. The TCP/IP stack has largely become a monoculture so RINA can only be of practical use if it can work *with* TCP/IP, at least initially. There's more than one way that RINA can coexist with TCP/IP.

In fact, it should be even easier to adopt RINA than to follow the IETF's recommendations and transition from IP version 4 to IP version 6. That's a difficult transition at best. IPv6 was not designed for backward or forward compatibility with its predecessor. But they occupy the same place in a fixed protocol stack. So the usual strategy for coexistence is for systems to run dual stacks. They try to connect using IPv6, and if that fails, they try v4. So every node needs to stay on IPv4 until v6 can reach everyone you need to reach. That in turn forces the issue of preserving the availability of IPv4 address space, so there isn't much incentive to migrate. It is a strategy for atrophy.

RINA provides more options for its phased adoption. It is not fixed to one place in the protocol stack, so implementations can be flexible. It treats TCP/IP as a sort of limited DIF. RINA thus can be applied *below* TCP/IP, to provide a backbone network that provides service to TCP/IP hosts, or to join IP networks together. RINA can also be used *above* IP, using existing IP links as a transport medium. And RINA networks can be *gatewayed* to TCP/IP networks, translating at least some applications between the two.

Adoption of RINA, once it becomes available, could thus make use of all three of these, gradually supplanting TCP and especially IPv4. A backbone network could be built using RINA, supporting both types of upper layers. And RINA-native applications could be developed and rolled out. Since RINA does not depend on globally-unique IP addresses, a single IP address could function as a gateway to a RINA network in which applications communicate using names.

The first stage of adoption is to phase out the use of IP addresses within the application layer. RINA considers NAT to be a normal function – addresses are a local function within the black box – so anything that violates that principle is going to be harder to migrate. So it's possible to start learning the lessons that RINA did and applying them within TCP/IP. This will help existing networks while taking a step forward. And, of course, stop wasting time and money on IPv6, which simply does not solve any of IP's real problems.

## Summary of benefits

It can be seen that RINA offers a number of benefits compared to TCP/IP. These include improved scalability (the ability to efficiently support larger networks) and security (both privacy of communications and protection against attack). It addresses the "3M" challenges of mobility, multicasting and multihoming. It provides a standard mechanism for application development. It can moot the neutrality issue by providing QoS options without allowing deep packet inspection or even making the application visible to an underlying network. It provides an easy adoption path for IPv4 users. And it does all this with an elegant simplicity that will facilitate lower cost,

higher-efficiency implementations. By returning to the fundamentals and recognizing the patterns in network architecture, RINA promises to move networking to the next level.

[As of this writing, RINA prototypes are still in the design phase and not yet ready for deployment.]