

# **Network Architecture: Principles, Guidelines**

**Shivkumar Kalyanaraman**

**[shivkuma@ecse.rpi.edu](mailto:shivkuma@ecse.rpi.edu)**

**Web Search Keywords: “Shiv RPI”**

# **Network Architecture: Principles, Guidelines**

## **Shivkumar Kalyanaraman**

|   |    |
|---|----|
| What is Network Architecture ? .....                                  | 3  |
| Introduction.....   | 3  |
| Goals .....   | 4  |
| Technological Building Blocks .....                                   | 5  |
| Evolution or Revolution?.....   | 9  |
| Tradeoffs in Cross-Layer Performance Optimizations .....              | 12 |
| Function Decomposition, Organization and Placement .....              | 14 |
| Design Principles .....   | 16 |
| The End-to-End (E2E) Argument.....                                    | 17 |
| Performance, Completeness and Correctness .....                       | 18 |
| State, Signaling and Fate-Sharing.....                                | 19 |
| Routing and E2E Argument: A Contradiction?.....                       | 20 |
| Application to QoS, Telephony and ATM networks .....                  | 21 |
| Active Networking.....  | 23 |
| Evolution of the End-to-End Argument: Tussles and Layered Naming..... | 24 |
| Summary .....   | 26 |
| References.....   | 28 |

## What is Network Architecture ?

Main Entry: **ar·chi·tec·ture**

Pronunciation: 'är-k&-"tek-ch&r

Function: *noun*

**1** : the art or science of building;...

**5** : the manner in which the components of a computer or computer system are organized and integrated

--- Merriam-Webster English Dictionary

### *Introduction*

Congratulations! You have been given the coveted job of a network systems architect. But, you wonder: "What does it *really* mean to "architect" a network system?"

Think of what a *home* architect does. She first understands her customers' desires for a residence and incorporates them to come up with "designs" and detailed "plans". Once these designs are finalized, they will be implemented and executed by a builder. So, architecture deals with the design and organization phase, and is distinct from implementation.

Just like a home architect, a network system architect would go through a *process* of stepping back and thinking in a *top-down* fashion what needs to be done to build the network system. The steps in the process can be summarized as follows:

1. Define the *goals* of the system.
2. Understand the *technological building blocks* and the cost-performance tradeoffs they present. Then, construct a system design by combining these building blocks to minimize overall system cost and meet system goals.
  - a. Technological building blocks are subject to *exponential cost-reduction trends* that affect the *relative* cost structure of the system components, or enable new cheaper substitute components. The system design must be roughly consistent with these trends for the intended lifetime of the design.
  - b. Technological building blocks may enable new performance dimensions that can help reshape system goals (eg: packet-switching vs circuit switching, peer-to-peer overlays vs regular internet).
3. Manage the architectural *complexity* and make it *evolution-friendly* using ideas like modularity, layering and stable interfaces.
4. *Minimize* or carefully architect any *cross-layer designs* or implementation *optimizations* to support long-run evolution, stability and inter-operability of interfaces.

5. *Decompose* system functions, *organize* these functions into data/control/management planes and *place functions* appropriately in the distributed system, guided by design principles like the *end-to-end* (E2E) principle.

We will look at each of these aspects in detail.

### **Goals**

First, the network architect needs to define the *goals* of the system (i.e. what primary function and capability should the system achieve?). Goals give clear focus and discipline the process of system design. This seems easy enough, but goals are often vague or not stated explicitly. Also, the architect needs to appreciate that inappropriate goals or poor adherence to goals increases the risks of architectural confusion.

For example, the basic goal of the Internet is to provide "connectivity" or a "best-effort virtual link" abstraction. Given this goal, one could then drill down to understand and define what "connectivity" or a "best-effort virtual link" really means in terms of protocol mechanisms. An example of straying away from this focus includes trying to over-optimize best effort virtual link performance and attempt to provide "type-of-service" (i.e. weak quality-of-service) capabilities. These attempts led to niche ideas such as load-sensitive routing (in ARPANET) and TOS-based routing that have either failed or have not been adopted broadly. The TOS-field has now been salvaged by the differentiated services (DS) group in IETF, once the goals were clarified for its use.

Goals matter! Small variations in goals can have dramatic implications for the architecture. For example, small variations in the goals of the IP multicast model led to dramatically different architectures and complexities. Similarly architecting a network to deliver "best-effort" virtual links is very different from a network that supports a "leased line" model for virtual links that in turn is different from a network architecture supporting a rich *variety* of quality-of-service (QoS) capabilities. These goals led to the development of the TCP/IP-based Internet, the telephony system and ATM networks respectively. A modest shift in high level goals (in particular, the nature of virtual links supported) is the cause for the dramatic divergence in these architectures.

In summary, it is important to think through the real meaning and implications of particular goals. In particular, how exactly are these goals different from alternative goals? How different or desirable are the alternative goals in terms of their architectural implications? Simplicity and clarity of the goals have the potential to discipline the subsequent architectural thinking.

## *Technological Building Blocks*

At the high-level, system design involves the following steps:

1. Decide what you want (goals, functions, attributes of the system).
2. Look at what you have to build the system (technological building blocks).
3. Put the building blocks together to achieve your goals while minimizing cost.

Once the system goals are established (as discussed in the previous section), the architect must consider the *technological building blocks* available and the cost-performance tradeoffs they offer. Technological building blocks include hardware and software components. In addition, various *conceptual* building blocks in networking include multiplexing, indirection, virtualization, randomization etc. These building blocks are put together to build the system.

Two key aspects of building blocks are: a) cost-performance behavior and b) complementarities with other building blocks. A system design combines complementary building blocks in different proportions to minimize overall costs without compromising system goals. Different sets of components can serve as mutual *complements* in a design. Within a design, if a component is *cheaper* and can implement the same function, it can potentially replace the expensive component in the system design ("*substitute*"). This is possible only if the architecture is modular (where modules can be replaced easily).

For example, if backbone capacity is costly (as it was in 1970s-80s), it makes sense to incur the costs of QoS mechanisms to manage and allocate this capacity. When backbone capacity became *cheaper* and outstripped demand (eg: in late 1990s), and applications were better able to tolerate performance volatility, the rationale for complex QoS mechanisms was questioned. In other words, cheaper capacity and adaptive applications are potential substitutes for complex QoS management mechanisms. This example also shows that cost-performance tradeoffs change over time and could dictate changes in system design over time.

Another example involving capacity tradeoffs is packet-switching vs circuit-switching. Packet switching was developed in the early 70s (when capacity was costly) and involved the idea of breaking up *demand* into chunks ("packets") rather than breaking up end-to-end *capacity* into chunks ("circuits"). Packet switching would allow packets arriving in a burst to be queued at a bottleneck. Observe that capacity is assumed to be costly. Instead of peak provisioning of costly capacity (as in circuit switching), packet switching only needed to ensure that the *average* capacity exceeded average demand, leading to statistical multiplexing gains. At the core of the Internet, with the recent availability of *cheap* capacity, the rationale for packet switching is slipping, and circuit switched designs are re-emerging (eg: DWDM and optical networking). Moreover, the availability of rapid re-provisioning tools in optical networks (eg: using G-MPLS) reduces the costs of over-provisioning seen in original circuit switching. While these new techniques may apply at the backbones of the Internet, they cannot replace the entrenched packet switching paradigm end-to-end.

Similarly IPv4 is an architectural choice that is *entrenched* everywhere on the internet. IPv6 is an upgrade to IPv4 that has not been successful in gaining a foothold yet despite enough mechanisms that allow incremental deployment. The original motivating problem of scarce IPv4 address space has been indefinitely deferred by using a mix of savory and unsavory mechanisms. Subnet masking and classless inter-domain routing (CIDR) allow efficient use of a given address space. Private IPv4 addresses using network address translation (NAT) and leasing of IPv4 addresses using DHCP allow the dynamic sharing of the scarce public IP address space.

While network architects are sympathetic to DHCP as a valid way to manage scarce public address space, they tend not to like the NAT technique. This is because several legacy applications and IP security techniques depend in myriad ways upon the end-to-end IP address, and a NAT box must go beyond just IP-address translation and translate the IP addresses embedded in applications (i.e. violate layering). Such layer violation and re-writing of higher layer information is incompatible with end-to-end security and encryption. However, the persistence of IPv4 and NAT mechanisms illustrates the impact of complementarities: it is hard to change a set of complementary mechanisms that are widely and deeply entrenched.

Another phenomenon occurring in packet-switched networks is congestion. Though congestion happens due to overload at local bottlenecks, it is ultimately experienced by end-to-end flows. Congestion control involves careful design of end-to-end mechanisms to reduce overload at bottlenecks. If backbone capacity becomes cheap, congestion may *shift* from the backbone to the access network. The *end-to-end* congestion problem does not go away (i.e. is not substituted by cheap backbone capacity). However, if *end-to-end* capacity can be provisioned rapidly in the future (eg: through signaling and resource reservation protocols that are *used* widely), then the pure rationale for end-to-end demand management mechanisms may change. In other words, a single link capacity is not an effective substitute for end-to-end congestion control, though rapidly provisioned end-to-end capacity could be a substitute.

So far we have discussed the complementarities and substitution of building blocks in modular architectures. Even though we could substitute components in a system, the performance impact of a single component change on the entire system may vary wildly. In particular, the impact depends upon the proportion of the system *unaffected* by the change. Amdahl's law, named after computer architect Gene Amdahl [Amdahl], says that if a proportion P of a system is speeded up by a factor s, the overall speedup is

$$S = 1 / [(1 - P) + P/s]$$

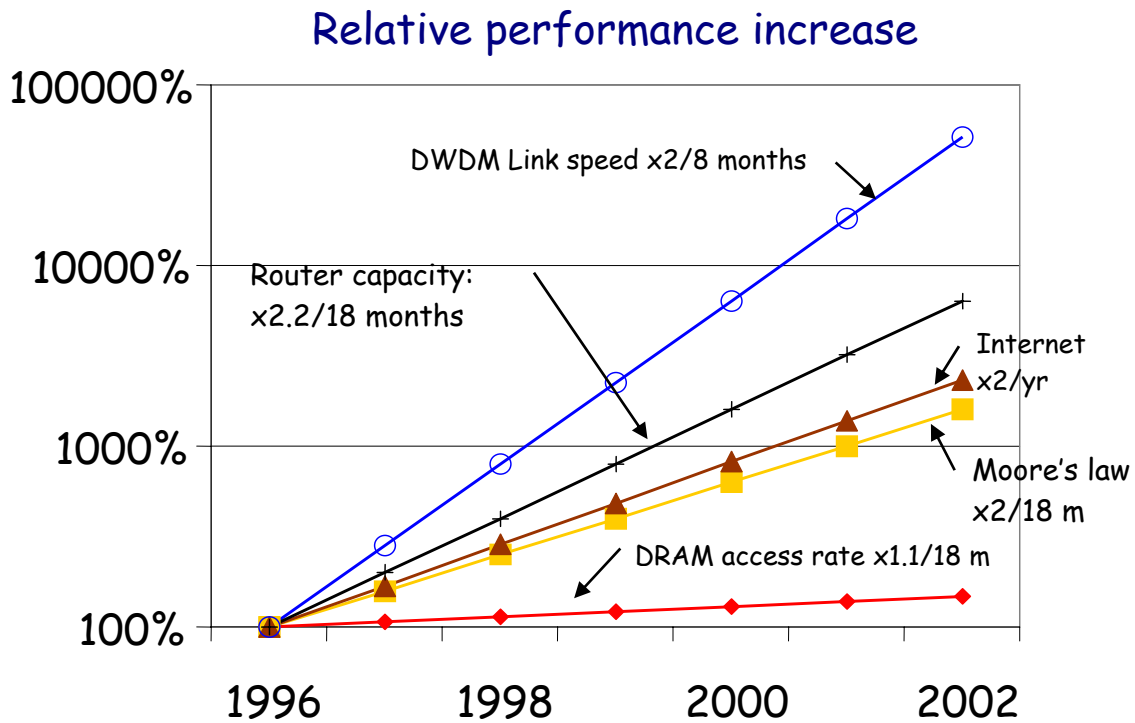
Even if s is infinity, the denominator is (1 - P). For example if P is 0.5 and s is infinity (i.e. half the system is speeded up), the total system speed up is  $1 / (1 - 0.5) = 2$ . Think about what this means: even if half the system is *infinitely* speeded up, the net effect on the *overall* system is only a factor of 2! In effect Amdahl's law is a demonstration of the law of *diminishing returns*, and also points to the value in *targeting* system improvements better to maximize overall speedup. In particular, you get a larger bang for

the buck if identify and speed up the common case or costliest sub-components of the architecture.

Sometimes, system design can be driven in the opposite direction, i.e. in a bottom-up manner: instead of pre-setting goals for the system, the availability of new building blocks and their cost-performance tradeoffs may enable a completely new design. In particular, lower performance at lower costs may be an attractive option if the cost reduction is substantial, and newer attributes (eg: size, portability) are enabled. Such bottom-up system design innovations driven by changes in cost-performance attributes of technology are termed "*disruptive*" innovations ([3]) and have occurred often in the past (eg: PC vs mainframe, laptop vs desktop). As will be discussed in the next section, disruptive innovations also often change the interface presented to users to better reflect the radical shifts in relative cost-performance attributes of its components.

Another key difference between the building blocks used in the design of a home and the technological building blocks used in networking is that the latter change *rapidly* and are subject to powerful economic "laws" i.e. *exponential* cost reduction trends (eg: Moore's law, Metcalfe's Law). Specifically, the cost of hardware drops significantly (50%) and capabilities increase every 18 months; optical networking innovations (like DWDM) lead to increases in optical fiber capacity at an even faster rate (every 12 months).

When costs of key system components change rapidly and other components' costs do not, then within a fairly short period the *relative cost structure* of the system components may change substantially, demanding changes in system design. In a good design, the *major trends* in technology cost evolution are identified, and the system may be designed in a modular fashion to allow upgrades in the components that are affected by such exponential cost reductions.



**Figure 1:** Technology Trends Affecting Network Architecture (from Nick McKeown, Stanford)

For example, high-speed router design is in essence an optimization problem involving power, space and cost constraints [4] while providing switching capacity that demonstrates cost reduction trends at a rate faster than Moore's Law, but slower than optical fiber capacity (see **Figure 1**). Moreover, router design has to be done while using components (eg: computing, memory) that trends at a rate equal to or slower than Moore's law. This *long-run trend rate mismatch* has to be solved through careful architecture and implementation. In this specific case, it is obvious that processing and memory access are key bottlenecks because of a severe trend rate mismatch and must be carefully optimized.

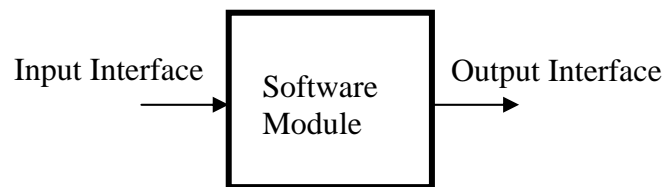
In summary, technological building blocks are used to build the system to meet the design goals. The rapidly changing costs of such building blocks imply that systems should be built in a modular fashion to accommodate substitution of building blocks. Sometimes technological progress may lead to building blocks that do not fit the pre-existing modular structure and could be combined with other complementary building blocks leading to a new design. The next section deals with the idea of modularity and interfaces further.

## ***Evolution or Revolution?***

We have seen that network protocols are complex, distributed pieces of software. Three key features of network protocols are: system complexity, distribution of components, and the need for evolution (i.e. protocol upgrades). To deal with *complexity* and *evolution*, protocol designers use well-known ideas from software design: abstraction and modular design.

After making assumptions about the path of technological progress (eg: Moore's law, software development costs, the economics of system evolution etc) the architect needs make her architecture *evolution*-friendly consistent with these assumptions. An evolution-friendly architecture will allow the system to incrementally evolve by incorporating trends such as Moore's law without fundamentally disrupting its users. Abstraction, modularity, layering and *invariant* interfaces are examples of mechanisms that enable evolution.

By *abstraction*, we mean that a subset of functions is carefully chosen and setup as a "black-box" or *module* (see **Figure 2**). The module has an *interface* describing its input/output behavior. The interface typically outlives the *implementation* the module in the sense that the technology used to implement the interface may change often, but the interface tends to remain constant. Modules may be built and maintained by different commercial entities (often also at competition with each other). The software modules are then used as building blocks in a larger design. The division of a complex system into functions and the placement of functions into modules, and the definition of interfaces is a core activity in software engineering.



**Figure 2: Abstraction of Functionality into Modules**

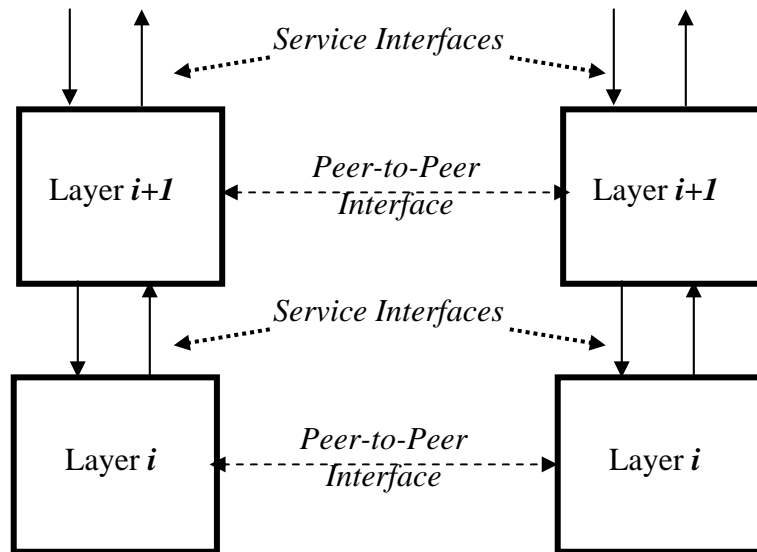
Protocols have an additional constraint of being distributed. To deal with distribution of components, protocol designers follow additional design principles and guidelines. First, the software modules discussed above have to communicate with one or more software modules *at a distance*. Such interfaces across a distance are termed as "*peer-to-peer*" interfaces; and the local interfaces are termed as "*service*" interfaces (**Figure 3**). Since protocol function naturally tend to be a sequence of functions, the modules on each end are organized as a (vertical) sequence called "*layers*".

The architecture specifies the decomposition into functional modules (including layers), the semantics of the individual modules and, the syntax used to specify the protocol. The

set of modules organized as layers is also commonly called a “*protocol stack*”. The concept of layering is illustrated in **Figure 4**.



**Figure 3: Communicating Software Modules**



**Figure 4: Layered Communicating Software Modules (Protocols)**

Once users adopt a system, they prefer its interfaces not to change (i.e. remain *invariant*), but expect new technology to be incorporated within layers while maintaining the stability and invariance of interfaces. However, this may not always be possible if building blocks do not evolve as planned. For example a new building block may require relaxing some aspects of the interface or performance expectations of applications.

The next best thing to invariant interfaces is *slowly evolving* interfaces. Simple economics reveals why slowly evolving interfaces are important: often the entire system is not built by a single organization. The complementary building blocks are implemented by different organizations, and the complementarities are defined by the interfaces. Interoperable, invariant interfaces therefore allow massive division-of-labor

and specialization while giving users and implementers some confidence that the entire package of complementary components will work together. This leads to rapid technology adoption, proliferation, economies of scale and unit cost reductions. The resulting profits fund the next generation of system development targeted at new capabilities valued by users. Disruptive innovations that are driven by shifts in technology trends shift the architectural interface presented to users (one more reason why they are labeled “disruptive”) and enable completely new applications and open new markets [Eg: think about the handheld vs PC vs mainframe in terms of applications and markets.]

Interfaces can also be *open* or *closed* and can be either *public* or *private*. Different combinations lead to different rates of adoption, proliferation and distribution of profits. Java is an example of an open, private interface. Microsoft Windows API and the Intel processor instruction set are examples of semi-open, private interfaces. TCP/IP and Linux platform are examples of open, public interfaces. Open, public interfaces offer the lowest costs and allow rapid proliferation. The standardization of key internet protocol interfaces (especially TCP/IP and BGP-4 inter-domain routing) in an open, public manner by the Internet Engineering Task Force (IETF) led to such rapid proliferation. Both TCP and BGP have evolved rapidly, but their interfaces have evolved slowly. Moreover, the changes have been incrementally deployed without requiring upgrades everywhere in the network. Indeed it is a necessity today that new protocol enhancements be *incrementally deployable* to facilitate smooth evolution. However, as certain protocols of the Internet become well entrenched and harden, even incremental deployability does not guarantee adoption since too many locations need to be upgraded and retraining costs are perceived to be very high (eg: the troubles experienced by IPv6 deployment).

Long-term architectural and interface design decisions are often made at the beginning of revolutions, but must *explicitly plan for evolution*, the more common phenomenon. It is often the case that technological progress provides opportunities for *revolution*. By revolution we refer to new “disruptive” systems and architectures that become possible because of dramatic changes in relative costs of technological building blocks or fundamentally new capabilities in building blocks. For example, the packet switching paradigm and its accompanying end-to-end design principle is a disruptive change compared to the legacy of TDM-based circuit switching. This change was enabled by the possibility of complex processing at end-systems combined with the possibility of queuing and data-plane packet processing at routers. On the same note, another potential disruptive change, “active networking” which envisaged code in packets and execution of code in routers did not succeed due to the constraints on complex processing in the data plane, combined with security and administrative control concerns.

In summary, modular designs allow evolution (through substitution of components enabled by technological advance) even if it was conceived as a revolutionary design. Modularity and interfaces should be designed in such a way that the technological changes will largely affect the modules and not the interface (invariant interfaces). If the advances must be adopted and affect the interface, it is best to slowly evolve the interface minimizing the impact on complementary stakeholders.

## *Tradeoffs in Cross-Layer Performance Optimizations*

There is always a tension between *short-term* performance optimization and *long-term* stability of architectural interfaces. It is usually possible to produce more efficient code by jointly optimizing the multiple layers. Such optimization could be done either at the design stage (called "*cross-layer design*") or at the implementation stage (called "*integrated layer processing*"). Optimizations that are done after the implementation and deployment stage are termed as "*layer violations*".

Cross-layer design, which proposes optimization across layers at the design stage, has become a hotly debated topic in the area of wireless networks [2], since new methods at the physical layer may provide tantalizing performance gains, if the interfaces to higher layers could be modified. However, at the other extreme, such approaches run the risk of not allowing evolution.

One example of a successful cross-*component* optimization in wireless involves modulation and error-control coding: components that are often designed together to provide adaptive, robust and high performance integrated functions (eg: turbo-coding, space-time coding, adaptive modulation/coding). However, since both these components are considered part of the physical layer and integrated by a single vendor this does not pose the usual problems of cross-*layer* design. But, an attempt to optimize wireless QoS at base stations by integrating layer 3 mechanisms (queuing, scheduling, buffer management) with layer 2 mechanisms (differential backoff or priority access in MAC) and layer 1 mechanisms (priority or classes within PHY modulation/coding/transmission) is a cross-layer design that is hard to evolve *even* if the integration and upgrade is handled by a single vendor.

One method attempting to strike a balance between the benefits of layering and the gains of cross-layer cooperation is "*application layer framing*" (ALF) [7]. ALF was proposed in 1990 and has been successfully used in the real-time transport protocol (RTP) used widely for VoIP and multimedia applications. ALF is aimed at providing flexibility and optimization within a layered framework. ALF gives control of framing decisions to the application layer. The transport layer provides a partial set of header fields that can be extended by the application. The real new "layer" here is an integrated transport/application layer. RTP provides some basic fields and functions, but the application codecs can extend RTP headers in proprietary manners to create a final header to be used in the multimedia application.

Integrated layer processing (ILP) is another approach that distinguishes between architecture and implementation. ILP suggests that while the architectural thinking about functions can be done in terms of layers, the implementer could have the flexibility to avoid serial processing and rather implement multiple layers in one or two integrated loops. This approach would be appealing if the layers in question are designed and implemented by the same organization. However, the authors of the ILP approach [7] do

express concern about the maintainability and ongoing utility of such customized implementations.

Cross-layer optimization may also be proposed after an architecture is in place and a new function needs to be inserted into the stack urgently (for economic or expediency reasons) that needs to make assumptions about higher layer behavior. Such optimizations are referred to as "*layer violations*", and are frowned upon by network architects. In particular, such designs need to be updated whenever applications change.

For example, TCP performance enhancement proxies (PEPs) on 3G wireless links involves the use of layer 2/3 functions that are TCP aware, and may also modify TCP header fields in packets that pass through the proxy. Deep packet inspection engines at edges of enterprise or ISP networks similarly attempt to optimize application performance by looking at application-layer headers and intelligently mapping packets to the layer 3 service provided by the network. Network Address Translation (NAT) is a middlebox function that maps private IP addresses to public IP addresses and allows the multiplexing of the scarce IPv4 address space. NAT unfortunately requires layer-violating interactions with higher layers that have strong dependencies on the IPv4 address (such as TCP, FTP, IPSEC tunnels).

Despite stiff opposition, some layer violations that have survived the test of time (eg: NAT, 3G TCP performance enhancement proxies (PEPs)). Usually, any design that survives the test of time tends to correctly identify system invariants that hold during the time period. For example the longevity of NAT and TCP PEPs is based upon the premise that the underlying technology/interface may not change as fast as generally thought (eg: IPv4 is not yet displaced by IPv6; wireless performance pitfalls for TCP persist). However, these very violations may create impediments for evolution and future elegant design (eg: emerging peer-to-peer applications such as Skype need to use servers for discovering peer IP addresses and NAT discovery hacks to go seamlessly through NAT boxes). To the extent that applications tolerate such impediments, these optimizations tend to be long-lived.

In summary, invariant or slowly varying interfaces *often* tend to override short-term performance optimization considerations. The reason is that, often, such short-term performance gains tend not to be order-of-magnitude improvements and are quickly overtaken by long-term economic gains and lower system costs due to rapid proliferation. Moreover, the relentless march of Moore's law that would make a layered design match the optimized performance of a cross-layer design just by replacing lower layers with order-of-magnitude faster hardware. In summary, an architect should think twice before locking in cross-layer optimizations or layer-violations for the long-term. At the same time, some layer-violation features like NAT have proven to be long-lived because they provide a critical intermediary service and rely on the existence of relatively stable protocols and interfaces (eg: NAT relies on the popularity of IPv4). Also, the perceived costs of the alternatives to such layer-violation designs have been high (eg: the perceived costs of IPv6 deployment).

## ***Function Decomposition, Organization and Placement***

As discussed earlier, network protocols are complex, distributed pieces of software. The three key features are: system complexity, distribution of components, and the need to accommodate evolution (i.e. protocol upgrades). So far, we have discussed issues to handle complexity and evolution: modularity, interfaces and layering. We now consider guidelines for how the networking system is *decomposed* into functional modules, how these modules are *organized* into groups or "planes", and finally how they are *placed* at different points in the network. Since a lot of work in explicit architecting goes into function placement, we will sometimes interchangeably use the terms "network architecture" and "placement-of-functions".

Network protocol functions are placed at different distinct nodes (eg: end-systems, edge/core routers, servers, clients, peers, overlay nodes etc). Efficient coordination between different functions usually involves the maintenance of shared information across time (called "state") at various nodes and in packet headers.

What architectural guidelines are available for an architect to organize such functions and to think about these issues?

The broad organization of functions into planes is dictated by *time- and space-scales* (see Rexford et al [5] for a survey and critique):

- **Data-plane** functions (eg: congestion control, reliability, encryption) are those that happen at small time-scales and involve substantial packet handling operating at the speed of packet arrivals. For example, the data plane performs packet forwarding, including the longest-prefix match that identifies the outgoing link for each packet, as well as the access control lists (ACLs) that filter packets based on their header fields. The data plane also implements functions such as tunneling, queue management, and packet scheduling. In terms of spatial scales, the data-plane is local to an individual router, or even a single interface card on the router.
- **Control-plane** functions (eg: routing, signaling, name resolution, address resolution, traffic engineering) happen at a longer time-scale and facilitate data-plane functions. The control plane consists of the network-wide distributed algorithms that compute parts of the state required for the data plane. For example, the control plane includes BGP update messages and the BGP decision process, as well as the Interior Gateway Protocol (such as OSPF), its link-state advertisements (LSAs), and the Dijkstra's shortest-path algorithm. These protocols compute the forwarding table that enables data-plane packet forwarding.

“Signaling” protocols (eg: MPLS, ATM PNNI-based signaling, telephony/X.25/ISDN/frame-relay signaling like Q.2931b) that associate global identifiers (addresses) to local state (eg: labels, resources) fall into this category. End-to-end signaling (eg: SIP, TCP connection setup, IPSEC session setup) are

also control-plane functions that setup data-plane enabling state. The Domain Name Service (DNS) is also a control plane function that maps names to addresses and enables end-to-end data plane activities. Control-plane functions may be *data-driven*, i.e. triggered by a data-plane event (eg: ARP, DNS), or be purely *control-driven* and operate in the background (eg: OSPF).

Peer-to-peer building blocks like distributed hash tables (DHTs) and I3 blur the lines between control and data plane. They involve control plane activities to determine the mapping from a key to a value, but this resolution also ends up at the location where the value is stored. Commercial systems like Napster, Kazaa, Skype incorporate innovations primarily in the control plane.

- *Management-plane* functions (eg: SNMP, active probes, tomography) facilitate monitoring, management and troubleshooting of networks and work at an even larger time-scale than control-plane functions. The management plane stores and analyzes measurement data from the network and generates the configuration state on the individual routers. For example, the management plane collects and combines SNMP (Simple Network Management Protocol) statistics, traffic flow records, OSPF LSAs, and BGP update streams. A tool that configures the OSPF link weights and BGP policies to satisfy traffic engineering goals would be part of the management plane. Similarly, a system that analyzes traffic measurements to detect denial-of-service attacks and configures ACLs to block offending traffic would be part of the management plane.

In today's IP networks, the data plane operates at the *time-scale* of packets and the *spatial scale* of individual routers, the control plane operates at the of timescale of seconds with an (optional) incomplete view of the entire network, and the management plane operates at the timescale of minutes or hours and the spatial scale of the entire network. Recent architectural rethinking involves proposals to shift or consolidate functions from one plane to the other (eg: Rexford et al [5]). For example, some of the decision functions from the control plane (eg: BGP routing) could be moved to the management plane to enhance the stability, responsiveness and functionality of routing. Another architectural redesign example (BANANAS Traffic Engineering Framework [9]) proposes to avoid the function of explicit control-plane signaling to achieve a subset of explicit path routing capabilities seen in complex architectures (eg: MPLS networks). This change will allow richer traffic engineering capabilities with minimal, incremental upgrades in legacy connectionless protocols (eg: OSPF, BGP) that do not support signaling.

Once the functions have been broadly organized into planes, the detailed placement and coordination of functions and state is important. Guidelines for function placement include design principles like the "end-to-end" argument discussed in the next section). Together, this leads to a *qualitative* "decomposition" and "organization" of the abstract functions. Once such organization is decided, the architect can then get down to the detailed design of each function (eg: routing, congestion control, signaling etc). System

performance modeling in contrast refers to *quantitative* system abstractions that complement and guide the qualitative architectural decision-making process.

There are different *locations* in which functions can be placed in the network. The basic divisions are "*end-systems*" (i.e. internet hosts) and "*network-elements*" (i.e. routers, switches etc). Certain control-plane functions and their associated state variables (eg: routing) are placed in layer 3 and largely in the network elements; hosts usually have simple default routes. At the same time, per-flow functions like reliability are placed at layer 4, and in the end-systems; routers do not carry any per-flow state. This contrasts with telephony-inspired virtual circuits (VC) in networks such as X.25, ATM networks and frame relay, where switches carry state for each virtual circuit. One key motivation for the virtual-circuit model is to provide quality-of-service (QoS) guarantees to virtual circuits (though this was not an expressed goal in X.25).

Recently, the network nodes themselves have bifurcated into two categories: *edge* nodes and *core* nodes. Edge nodes form the boundary between the network and users, and tend to implement *processing-intensive* functions like traffic conditioning/policing and security related trust-boundary functions. Core nodes focus on high speed forwarding of packets with minimal additional processing. Core router designs have to keep up with the growth in capacity of optical technologies. Routers have recently also bifurcated into hardware platforms that specialize in processing either control-plane information (eg: BGP, OSPF route processors) or implementing data-plane components (eg: ultra-high-speed forwarding). Typically the division of these functions and specialization also dictates where *state* information pertaining to these functions is placed and managed.

We have discussed a variety of locations in which functions and associated state can be placed. Different function placement decisions will also lead to different *degrees of complexity, evolvability*, and may or may not implement the "global" objective *correctly* or *efficiently*. The next section discusses design principles that provide guidance for the problem of function placement.

## **Design Principles**

*Those are my principles, and if you don't like them... well, I have others.*

Groucho Marx

The purpose of design principles is to organize and guide the placement of functions within a system. Design principles impose a structure on the design *space*, rather than solving a particular design *problem*. This structure provides a basis for discussion and analysis of trade-offs, and suggests a strong rationale to justify design choices. The arguments would also reflect implicit assumptions about technology options, technology evolution trends and relative cost tradeoffs. The architectural principles therefore aim to provide a framework for creating cooperation and standards, as a small "spanning set" of rules that generates a large, varied and evolving space of technology.

The Internet architecture, when conceived, was a break from the past due to its reliance on packet-switching and its proposal for a radically decentralized design. Another feature was its insistence to place intelligence in end-systems rather than inside the network with the goal of providing connectivity. This decentralization of function placement and concentration of intelligence end-to-end was guided largely by the end-to-end argument. After studying this argument and its implications in detail, we will overview current debates about how to extend the Internet architecture and new design principles that are emerging.

### ***The End-to-End (E2E) Argument***

The End-to-End argument was originally formulated by Saltzer, Reed and Clark[6]. Also referred to as the end-to-end (design) *principle*, it is a statement that *qualitatively* guides the debate about where functions or components of a distributed system should be placed. The end-to-end argument suggests that "*functions placed at the lower levels may be redundant or of little value when compared to the cost of providing them at the lower level.*"

In other words, a system (or subsystem level) should consider only functions that can be ***completely*** and ***correctly*** implemented within it. All other functions are best moved to the lowest system level where they can be completely and correctly implemented.

In the context of the Internet, the E2E argument implies that several functions like reliability, congestion control, session/connection management are best moved to the end-systems (i.e. performed on an "end-to-end" basis), and the network layer focuses on functions which it can ***fully*** implement, i.e. routing and datagram delivery. As a result, the end-systems are intelligent and in control of the communication while the forwarding aspects of the network is kept simple.

The end-to-end argument submits that even if the network layer *did* provide the functions in question (i.e. connection management and reliability), transport levels would have to add reliability to account for the interaction at the transport-network boundary, or, if the transport needs more reliability than what the network provides. Removing these concerns from the lower layer packet-forwarding devices streamlines the forwarding process, contributing to system-wide efficiency and lower costs. In other words, the *costs* of providing the "*incomplete*" function at the network layers would arguably outweigh the *benefits*.

End-to-end arguments arose from work on secure operating system kernels in the Multics project [8]; work on end-to-end transport protocols in LAN's and the Internet. Similar thinking by John Cocke et al on the role of compilers in simplifying processor architecture led to the RISC approach to processor architecture, which also suggests moving function from lower layers to more application-specific layers. An end-to-end argument is similar to the argument for RISC: it serves to remind us that building complex function into a network implicitly optimizes the network for one set of uses while substantially increasing the cost of a set of potentially valuable uses that may be

unknown or unpredictable at design time. A case in point: had the original Internet design been optimized for telephony-style virtual circuits (as were its contemporaries SNA and TYMNET), it would never have enabled the experimentation that led to protocols that could support the World-Wide Web. Preserving low-cost options to innovate *outside* the network, while keeping the core network services and functions simple and cheap, has been shown to have very substantial value.

The following subsections debate the interpretations and implications of this design principle, and further illustrate the value of its guidance in protocol design.

### ***Performance, Completeness and Correctness***

Does the end-to-end principle say anything about performance issues and cost-performance tradeoffs? Recall that the end-to-end argument focuses on the *distribution* of functions in a system (a.k.a. "*function placement*") subject to the constraints of correctness, completeness and overall system costs. The E2E paper says that, "sometimes an incomplete version of the function provided by the communication system *may be useful* as a performance enhancement." In other words, the argument indeed allows the existence of a *cost-performance tradeoff* and the incorporation of economic considerations in the system design.

However, it balances the goals of performance with completeness and correctness. In particular, it cautions that the choice of the placement of "*incomplete* versions of functions" inside the network should be made with prudence. Moreover, application-specific functions when placed inside the network may not be useful or even hinder the performance of *other* applications. In simpler terms, one should think twice (about performance and correctness) before placing redundant functions at multiple layers, and ask why they should be placed at lower layers at all.

The split of congestion control functions between TCP (an end-to-end transport protocol) and active queue management (AQM, implemented at bottlenecks) is an illustration of how the leeway in function placement can be used to *performance* reasons.

TCP implements an increase/decrease function, an RTT measurement function, a timeout and loss-detection function and a self-clocking function as its share of the Internet congestion control infrastructure. Intermediate bottlenecks are merely expected to drop packets during congestion, a function accomplished implicitly by the simplest FIFO queuing discipline and finite buffers. However, the body of work on active queue management shows that the performance of TCP/IP congestion control can be substantially enhanced through carefully designed AQM schemes. In particular, Kelly and Low have independently shown that TCP/AQM can be viewed as implementing a distributed primal-dual algorithm in the quest of a global utility optimization problem. The end-to-end argument does allow the judicious placement of partial functions like AQM to enhance the TCP congestion control function.

## *State, Signaling and Fate-Sharing*

One issue regarding the "incomplete network-level function" is the degree of "state" maintained inside the network. "State" is a term used to denote "stored information", i.e. information transferred from one function to another over time. Just like "functions" are "placed" at different locations guided by the e2e principle, "state" can also be similarly "placed" at different locations. A simplistic application of the end-to-end argument would suggest that state always be placed in end-systems instead of at network nodes. However, we illustrate that, for different functions (eg: per-flow reliability vs routing), state may be placed at different network nodes.

Connectionless networking protocols such as TCP/IP do not place "per-connection" or "per-flow" state *inside* the network. TCP connection state is maintained at end-systems only and IP does not maintain any per-TCP-flow state at routers. When we say that the network does not maintain per-flow signaling state (i.e. is *stateless* on a per-flow basis), an implication is that the network nodes do not need to notify each other as endpoint connections ("flows") are formed or dropped.

In contrast, explicitly signaled "*hard state*" for virtual circuits (eg: in telephony, ATM networks) needs to be explicitly setup, maintained and torn down by an out-of-band or distinct signaling protocol. While this state maintenance overhead is acceptable for building a network that offers quality of service (QoS), requiring resource and path reservations, it appears to be gratuitous for building a best-effort connectivity service. This is another implication of the end-to-end argument.

In contrast to per-connection state, routing protocols need to maintain routing state (i.e. routing tables) inside the network to facilitate forwarding. As we discuss in the next section, routing is a function that is placed inside the network. Even here, an implication of the end-to-end argument is to be cautious about the state that is distributed at remote nodes. Unlike signaled state, routing state is an example of "*soft state*." A distinctive feature of soft state is that it is maintained and refreshed as part of *normal* operation, i.e. in the background and not explicitly initiated. Failures are therefore handled as the common case operation rather than as exceptions. In contrast, repairing or maintaining hard state would require special exception handling mechanisms. Another example of soft state is RSVP-based QoS resource reservation which automatically expires unless refreshed. Similarly in ARP, the ARP table entries are timed out unless refreshed. Similar "aging" mechanisms are found in protocols like OSPF. No out-of-band or special protocol methods or messages are needed for tearing down such stale state. Inconsistencies are thus automatically retired locally, contributing to increased consistency of state in the distributed system achieved without explicit messaging for garbage collection. This "soft" nature of the state, i.e. its limited persistence and in-band refresh often simplifies the coordination needed to achieve robustness objectives.

Another state-related implication of the end-to-end principle is the concept of "fate-sharing". In a stateless network model, endpoints need not be aware of any network components other than the destination, first hop router(s), and an optional name

resolution service. Packet integrity is preserved through the network, and transport checksums and any address-dependent security functions are validated purely by the end-systems (i.e. the function is performed end-to-end). If state is maintained only in the endpoints, in such a way that the state can only be destroyed when the endpoint itself breaks (also termed "*fate-sharing*"), then as networks grow in size, the likelihood of component failures affecting a connection becomes unlikely. In particular, if a router fails en route, the end-to-end communication does not fail. The router does not share fate with the end hosts. In contrast, the OSI X.25 protocol gateways maintain hard per-connection state and end hosts shared fate with them.

Why does limited fate-sharing matter? If failures lead to loss of communication, because important state information is lost, then the network becomes increasingly brittle, and its utility degrades. However, if an endpoint itself fails, then there is no hope of subsequent communication anyway. Fate-sharing also has implications in a soft-state scenario. When failures of a part of the system compromise or destroy a portion of system state creating inconsistencies, the inconsistent state either times out or is repaired as a normal part of the protocol (eg: routing updates). Inconsistency does not persist.

In summary, the E2E argument suggests that per-flow state belongs to the transport layer that is logically resident in end-systems. Therefore, by simple layering principles, the state associated with the transport layer should not be placed at network nodes which do not implement transport layer functions. In those circumstances where state needs to be placed inside the network, soft-state that is maintained in the normal course of protocol operations and automatically self-destructs to avoid inconsistency is preferred over hard-state.

### ***Routing and E2E Argument: A Contradiction?***

In the previous section we discussed the implications of the E2E argument for state placement and fate sharing. While we saw that the E2E argument suggests that transport-level state be placed only at end-systems, we saw that the E2E principle does not dictate a completely stateless network layer. Though TCP/IP networks do not hold per-flow state, such networks indeed hold a lot of in-network state such as forwarding tables (i.e. routing state) useful for *all flows* and packets. Routing is a function that enables network-wide end-to-end connectivity and consistent per-hop forwarding. It operates directly on top of physical or virtual links between immediately adjacent nodes, and cannot assume multi-hop connectivity. Therefore the state required for routing and forwarding has to be placed at every router and the end-system (i.e. is fully distributed). In this sense, the placement of routing state is consistent with the end-to-end argument: "*... placing functions (and state) at the **lowest** system level where they can be **completely and correctly** implemented....*". Interestingly note that the routing state maintained at end-systems is trivial (eg: a default route) compared to the per-flow state at end-systems.

An interesting situation arises in source routing where the source specifies the partial or complete path that packets take through the network. In signaled architectures like ATM or MPLS, this source-route specification is mapped to local state information (eg: labels)

using the signaling protocol. In IP source routing and DSR routing in ad-hoc networks, a list of addresses is specified in each packet (potentially high overhead). Recent proposals involve compressing the per-packet dynamic state into a hash or a bloom filter, with optional soft state information computed by intermediate routers [9]. However, the hidden complexity in this method is that the source requires some “global” topology information to compute the source route.

This example illustrates a new location for state placement: in *packets* rather than network nodes or end-hosts. The path choice is dictated end-to-end, endearing it to end-to-end proponents. However, due to the inefficient coding of source routes, overheads in every packet, the requirement of global routing information at sources, and the lack of an incrementally deployable architecture, source routing (and multi-path routing) has not succeeded in connectionless networks. With the growth of multi-homing and availability of overlay paths on the internet, this situation may change in the future. Another proposal involving per-packet dynamic state is the DPS architecture for quality of service (QoS), where the state required to specify local QoS resources is specified in each packet rather than the use of out-of-band signaling, and using complex queuing mechanisms at routers.

### ***Application to QoS, Telephony and ATM networks***

Another dubious interpretation of the e2e argument is the claim that data networks such as the TCP/IP-based Internet conform to the end-to-end argument whereas the telephone network and QoS-oriented networks like ATM networks do not. This observation is typically used to draw the line between "Bellheads" and "Netheads". While we do not claim that the latter networks are in full conformance with the argument, the issue is more nuanced than it appears on the surface.

One of key issues in telephony and ATM networks is the need to provide Quality of Service (QoS) for each flow. A flow is either a stream of packets or for time-slots containing voice bits. The need for Quality of service (QoS) in a shared network implies a zero-sum game where bandwidth and delay resources need to be apportioned between flows and these assignments managed/policed. This need for apportioning resources requires a *trust boundary* between the "end-system" and the "network." Furthermore, there is a need for reservation "state" to either be installed inside the network, or carried dynamically in packet headers.

In the situation of ISPs providing Quality of Service (QoS) and charging for it, the trust boundary has to be placed inside the network, and not in end-systems. In other words, some part of the network has to participate in decisions of resource sharing, and billing, which cannot be entrusted to end-systems.

How should we view the end-to-end argument in the context of this QoS architecture problem? The end-to-end argument indeed allow placement of functions inside the network, to deal with the economic apportioning and trust model issues. Applications may be allowed to participate in the decision process, but the control of QoS apportioning belongs to the network, not the end-system in this matter. The differentiated

services architecture has the notion of the "network edge" which is the repository of a majority of these functions, clearly consistent with the end-to-end argument. In the "integrated services" or ATM network services, these functions are placed at both "core" and "edge" nodes. The line here is less clear since there might be more state and functions inside the network than necessary.

Telephone networks appear to have a diametrically opposite design compared to TCP/IP inter-networks. This is captured by the observation that telephone networks have "dumb" end-systems and "smart" switches. In other words, this implies a greater concentration of functions inside the network than might be prescribed by the end-to-end argument. Again, the situation is not that simple.

In the case of telephone networks, the original concentration of functions inside the network was driven by the technological and economic constraints of the age and limited/inflexible application requirements (voice).

Let us consider the economic and technological constraints. Unlike the Internet today, the service provider market for telephony was highly concentrated (AT&T monopoly in the US and PT&T monopolies in other countries). In other words, the control of network was concentrated. Automatic switches were brought in to replace human operators to scale the network. Technologically, the computer had not been developed in the early 20<sup>th</sup> century. Therefore, there was no possibility of software to power complex algorithms, leave alone doing this end-to-end. Even in the days when the first electronic switches and software were introduced in the network, their costs were so high and would not be economical for end-system deployments.

In the language of the end-to-end argument, the lowest system level where the telephony "protocol" functions could be correctly implemented, consistent with economics and available technology, was at the switch. Therefore there is no contradiction with the end-to-end argument if these technological and economic clauses are applied. However, as computers became available and the protocol concepts became well developed, the economics and technological backdrop changed. These changes fit newer data applications, but still did not fit high-quality voice telephony services for a couple of decades. Since voice was the big revenue generator for telephony, there was no pressure to revise the core telephony architecture.

The real Bellhead-vs-Nethead debate came during the development of "integrated services" (ISDN and B-ISDN) by the telephony service providers seeking a smooth upgrade to networks supporting voice, video and data services. The Nethead group valued legacy voice applications less, pointing to the surge of data applications, industry fragmentation, and pace of technology improvements. The Bellhead group focused on compatibility and seamless transition of their existing residential and business voice, and private data line (eg: leased line, frame relay) customers onto a single network (the ATM network). ATM network technology was criticized as being philosophically incompatible with the end-to-end principle since it tried to be a one-size-fits-all solution and due to its high complexity. The reality that played out largely vindicated the Netheads.

Overcapacity caused by IP/MPLS-based optical network deployments, wireless phones and Voice-over-IP (VoIP) protocol adoption created strong substitutes for the legacy voice carriers. The price of voice calls plummeted, but the data network revenues have not ramped up equally fast. The reasons for this shift were largely technological and economic in nature. The IP/MPLS architecture has complexity rivaling that of ATM network architecture, but it avoids ATM-like features not relevant to IP.

The end-to-end argument did play an indirect role since it allowed the explosive adoption of the Internet that was an economic factor in this shift from voice to IP-based integrated services networks. Today, with the advent of voice-over-IP (VoIP), signaling protocols like SIP indeed operate end-to-end like TCP, and may or may not require QoS/resource reservations inside the network. The VoIP application today is far more flexible in terms of requirements than the original telephony application. However, for higher quality (“toll grade”) VoIP, some kind of aggregate QoS is provided by most operators and implemented using IETF protocols such as differentiated services.

In summary, the end-to-end argument has clearly promoted protocol deployment and the birth of new killer applications because new applications could be deployed quickly at end-systems. However, it is a misconception to rule out the entire telephony infrastructure as incompatible with this argument. A large part of the explanation for the perceived incompatibility has to do with factors such as economics, industry fragmentation, available technology and technological advances.

### ***Active Networking***

Active networking [9] allows programs or references to programs in data packets, with the intent that those programs be executed at points within the network. The purpose is to better customize the network service to application requirements. Does this idea violate the end-to-end argument?

End-to-end arguments separate design from implementation and execution, i.e., they suggest who should provide the code, not which box it should run on. Mechanisms like programmability that aid application-defined customization of services, even though the actual function is ultimately realized inside the network, are in line with the spirit of the end-to-end principle and application-layer framing (ALF) idea. However, unpredictable interactions between independently designed applications and independently acting users may pose a new problem that affects all users. Part of the context of an end-to-end argument is the idea that a lower layer of a system should support the widest possible variety of services and functions, so as to permit applications that cannot be anticipated.

The end-to-end argument can be decomposed into two objectives, which may conflict in this situation:

- Higher-level layers organize lower-level network resources to achieve application-specific design goals efficiently. There are no artificial barriers-to-entry for novel applications. (*application autonomy and innovation*).

- Lower-level layers provide only resources of broad utility across applications, while providing to applications usable means for effective sharing of resources and resolution of resource conflicts. The simplicity of core functions also promotes robustness. (*network transparency, robustness*).

In active networking we have the dilemma of increased application autonomy and the risk of reduced network transparency. To maintain the largest degree of network transparency, then, the end-to-end principle requires that the semantics of any active features be carefully constrained so that interactions among different users of a shared lower level can be predicted by a designer who is using the services and functions of that active layer. Lack of predictability thus becomes a cost for all users, including those that do not use the programmability features. Getting the semantics of active enhancements right is a major challenge, and wrong active enhancements are likely to be worse than none at all.

Thus, even though active network ideas are not ruled out by end-to-end arguments, the open challenge is to develop practical, high-impact examples of a sufficiently simple, flexible, and transparent programming semantics suitable for use in lower levels of networks. In particular, though the core of the internet is hard to change, active networking ideas could be useful in emerging niche areas like sensor networks where computation is several orders of magnitude cheaper than communications. Overlay and peer-to-peer networks may also provide a platform for active networking ideas to develop.

### ***Evolution of the End-to-End Argument: Tussles and Layered Naming***

The tremendous growth of the Internet has strained the limits of its architecture, protocols and design principles like the end-to-end argument. This section summarizes some of the key critiques of the architecture, key elements of counter-proposals and constraints on evolution.

The Internet has evolved from a context where all players viewed connectivity as its own reward to a new context involving multiple stakeholders whose interests do not necessarily align. These players include users, commercial ISPs, governments, intellectual property rights holders, content and service providers. Clark et al use the word “tussle” to describe the ongoing contention among parties with conflicting interests. The Internet is increasingly shaped by controlled tussles, regulated not just by technical mechanisms but also by mechanisms like laws and societal values. There is no “final outcome” of these interactions, no stable point, i.e. no static architectural model. For example, the desire for control by the ISP calls for less transparency, and we get deep-packet-inspection, application filtering, tiered access pricing based upon content and connection redirection.

Even within the technical realm, we have seen that the growth of the Internet is accompanied with the entrenchment of key interfaces (eg: IPv4) that makes large-scale evolution (eg: to IPv6) difficult, and middleboxes or intermediaries that offer expedient

indirection services (eg: NAT, overlays) proliferate. Similarly the loss of trust leads to the establishment of trust boundaries, security perimeters implemented by middleboxes like firewalls. The desire to improve important applications (e.g., the Web), leads to the deployment of traffic management middleboxes, caches, mirror sites, content delivery networks, and kludges to the DNS. The architectural issues raised include violation of IP semantics, making the Internet application-specific, and problem-specific hacks that complicates deployment of new applications. For example, new peer-to-peer applications like Skype have to detect and categorize the common types of NATs and firewalls in order to bypass them to reach centralized servers that maintain mappings of peers to public IP addresses.

One solution is that the tussles can be reduced or eliminated within the technical sphere using robust fault-tolerant designs that tolerate and overcome misbehavior, policy languages which allow the expression and alignment of interests, or game theory which allows the design of rules to eliminate unnecessary tussles. If tussles arise dynamically (in “run-time”) in heterogeneous venues and all the tussles have not been driven out of the system, the resolution goes well beyond technical solutions. However, some design principles can still be articulated that would frame the context for tussles and simplify future resolution of such tussles.

*Tussle Principle #1: design for choice or variation in final outcome – to accommodate future tussles rather than eliminate them.*

An example is to facilitate the expression of choice, value and cost in interfaces and allow flexible, negotiated binding to services.

*Tussle Principle #2: modularize the design along tussle boundaries, i.e. isolate tussles so that one tussle does not spill over and distort unrelated issues.*

Within tussle boundaries design principles such as the end-to-end argument may still be relevant. For example, the internet QoS architecture today differentiates between hosts, edge and core nodes. The edge nodes form a trust boundary, a.k.a. a tussle boundary. Within the scope of the network, there is a general tendency to move complex QoS customization functions to the edges, and allow the core nodes to focus on high-speed forwarding, while supporting a small set of abstract per-hop behaviors (PHBs).

A negative example of poor tussle modularization is today’s DNS design, which overloads DNS names with multiple semantics. This has led to spillover between the area of trademark and naming web objects. Similarly, although the Internet is now widely used by applications to gain access to services and data, it does not have a mechanism for directly and persistently naming data and services. Instead, both are named relative to the hosts on which they reside (i.e. in a host-centric manner). Using DNS to name data overloads the names and rigidly associates them with specific domains or network locations, making it inconvenient to move service instances and data, as well as to replicate them. The proposed rectification is to introduce modularity along tussle boundaries, i.e. introduce levels of indirection between them and avoid overloading

names or mapping mechanisms with needless semantics. Since the idea of introducing levels of indirection is an important general strategy, we elaborate it below.

A recent proposal by Balakrishnan et al [11] synthesizes a vast body of prior work in the area of redesigning the naming and addressing infrastructure of the internet. It proposes four layers of names: user-level descriptors (eg: search keywords, e-mail addresses), service identifiers (SIDs), endpoint identifiers (EIDs), and IP addresses or other forwarding directives. Their goals include (1) making services and data “first-class” Internet objects, i.e. they are named independent of network location or DNS domain and thus can freely migrate or be replicated (2) accommodating mobility and multi-homing (3) accommodating network-layer and application-layer intermediaries to be interposed as valid architectural components on the data path between communicating endpoints. Their proposed design principles in this space include:

*Principle #1: Names should bind protocols only to the relevant aspects of the underlying structure; binding protocols to irrelevant details unnecessarily limits flexibility and functionality.*

*Principle #2: Names, if they are to be persistent, should not impose arbitrary restrictions on the elements to which they refer.*

*Principle #3: A network entity should be able to direct resolutions of its name not only to its own location, but also to the locations or names of chosen delegates.*

*Principle #4: Destinations, as specified by sources and also by the resolution of SIDs and EIDs, should be generalizable to sequences of destinations.*

Observe that this proposal focuses on the naming architecture and limits its scope on purpose. The sheer size of the Internet’s installed router infrastructure renders significant changes to IP and routing infrastructure hard. However, if we sidestep issues that inherently involve routers (such as complete denial-of-service protection, fine-grained host-control over routing, and quality-of-service) there are many issues for which changes to IP would be irrelevant—and for which changes to the naming architecture would be crucial. Solving the broader list of problems in a practical, incrementally deployable way is still open as of this writing. It should be noted that these new proposals are still in the research prototype stage and time will tell which of these principles also dovetail well with the exigencies of real-world deployment and Internet architecture evolution.

## **Summary**

In summary, network architecture is a sub-branch of *distributed* software architecture. It is shaped by considerations from:

- a) software engineering (eg: complexity management, distributed components, evolution, function placement),

- b) systems design (eg: tradeoffs, technology, evolution, economics and functional objectives)

A key part of network architecture talks about the division and placement of functions in a distributed system to achieve the primary goal of networking: to implement the abstraction of a virtual link. Just like quantitative models give insight and guidance in the design of systems, network architecture and arguments like the E2E argument give qualitative guidance and insight in the design of networking systems.

The end-to-end argument in particular is significant, because it has guided a vast majority of function placement decisions in the Internet. This chapter has offered several examples to illustrate why it is important to interpret the E2E argument not in isolation, but along with considerations such as correctness, completeness, cost-performance tradeoffs, location of trust boundaries, technological and economic constraints.

It should be noted that end-to-end arguments are one of several important organizing principles for systems design. While there will be situations where other principles or goals have greater weight, an end-to-end argument can facilitate the design conversation that leads to a more flexible and scalable architecture. The new concepts of tussle and additional levels of indirection in the naming infrastructure are among attempts to evolve design principles to match current realities (multiple stakeholders) without impeding rapid evolution and application innovation.

The network systems we see around us today are the results of a mix of implicit or explicit architectural decisions.

## References

- [1] G.M. Amdahl, "Validity of the single-processor approach to achieving large scale computing capabilities," In *AFIPS Conference Proceedings*, Vol. 30 (Atlantic City, N.J., Apr. 18-20). AFIPS Press, Reston, Va., 1967, pp. 483-485.
- [2] V. Kawadia and P. R. Kumar, "A Cautionary Perspective on Cross Layer Design." *IEEE Wireless Communication Magazine*, Vol. 12, no. 1, February 2005, pp. 3-11.
- [3] C. M. Christensen, "The Innovator's Dilemma", *HarperBusiness Essentials*, ISBN 0060521996, January 2003.
- [4] I. Keslassy, S.T. Chuang, K. Yu, D. Miller, M. Horowitz, O. Solgaard, N. McKeown, "Scaling Internet Routers Using Optics," In *Proceedings of ACM SIGCOMM 2003*, Karlsruhe, Germany, August 2003.
- [5] J. Rexford, A. Greenberg, G. Hjalmtysson, D. Maltz, A. Myers, G. Xie, J. Zhan, and H. Zhang, "Network-wide decision making: Toward a wafer-thin control plane," In *Proceedings of ACM SIGCOMM HotNets Workshop*, November 2004.
- [6] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems* Vol. 2, No. 4, November 1984, pp. 277-288.
- [7] D.D. Clark, D.L. Tennenhouse, "Architectural considerations for a new generation of protocols", In *Proceedings of ACM SIGCOMM 1990*, Philadelphia, PA, 1990, pp. 200-208.
- [8] M. D. Schroeder, D. D. Clark, and J. H. Saltzer, "The Multics kernel design project," *Sixth ACM Symposium on Operating Systems Principles*, in *ACM Operating Systems Review*, Vol. 11, No. 5, November 1977, pp. 43-56.
- [9] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden, "A survey of active network research," *IEEE Communications Magazine* Vol. 35, No.1, January 1997, pp. 80-86.
- [10] H. T. Kaur, S. Kalyanaraman, A. Weiss, S. Kanwar, and A. Gandhi, "BANANAS: An Evolutionary Framework for Explicit and Multipath Routing in the Internet," In *Proceedings of ACM SIGCOMM Workshop on Future Directions on Network Architectures (FDNA)*, Volume 33, No. 4, Karlsruhe, Germany, August 2003, pp. 277-288.
- [11] D. D. Clark, J. Wroclawski, K.R. Sollins, and R. Braden, "Tussle in cyberspace: defining tomorrow's internet," *IEEE/ACM Transactions on Networking*, Vol. 13, No. 3, Jun. 2005, pp. 462-475.
- [12] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish, "A layered naming architecture for the internet," In *Proceedings of SIGCOMM 2004*, Portland, Oregon, USA, August 2004, pp. 343-352.