Architecting a Networked Application

David D. Clark

Version of Jan 12, 2005

1 Introduction

The primary function of a computer network is to provide communication among application programs running in end systems. Application builders, who are generally distinct from network builders, use this communication service without requiring a detailed knowledge of its inner workings.

Network builders depend upon design principles that they call "network architecture"; in layerist terms, network architecture is concerned with protocol layers up to and including the transport layer. In this low-level view of architecture, the application layer is a black box that is designed by someone else, the application builder. Application builders, of course, have their own abstractions. The most common abstraction is the "distributed system" model, in which the end application programs are components of a distributed application.

This section surveys a range of important design issues for network-based applications: application-layer servers, adversarial design, application-level naming and identity, robustness and resistance to attack, performance, economics, and user empowerment. These application-layer design considerations (which could be called "application architecture") are in part orthogonal to the architectural concepts at the network layers, although there are some areas of interaction.

The Internet already has a rich set of distributed applications; the most popular today are the Web, email, and peer-to-peer (P2P) computing. This section will draw on these applications for examples to illustrate the principles under consideration.

2 Application-Layer Servers

The dichotomy between network architecture and application design is being weakened by the rapid proliferation of middleboxes containing *application-layer servers* (ALSs). An ALS is logically part of one or more applications, but it is located "within" the network. From the viewpoint of the application designer, the ALS is part of the application, even though it executes on a machine remote from the users' end systems and not under their control. ALS middleboxes are interposed in some way in the communication path between the end points. An ALS may be specific to a particular application, or it may be a *common ALS* that provides a service available to any application. An ALS is sometimes called "middleware", but this term is overloaded and suspect.

The existence of ALSs is an apparent contradiction of the traditional end-to-end arguments. These arguments imply that all application-specific code should run in user end systems; the core of the network should be simple, application-independent, and focused on the basic network service of data carriage. However, this is an over-simplistic interpretation of the end-to-end arguments. First, from the network perspective, these applications are still end to end, because these ALSs are not part of the network. So the application may have many hops or stages in its design, but each of these seems to be end to end. From the application perspective, the design process has come to a more complex design: most of the popular applications today are not simply end-to-end. Email is not sent directly between user computers, it is sent from user computer to mail server to mail server to user computer. The Web, which was initially a simple system of end computers playing the role of client and server, is now rich with caches, proxies, and the like.

In fact, there are many possible good reasons for moving parts of an application from the user end-nodes onto ALSs. For example, an ALS can:

a) Stage content or cache data close to where it is needed, to reduce latency, reduce cross-network traffic, and improve performance (see Section 6.1).

b) Reformat or preprocess information before sending it over very slow links or to end-nodes with limited capabilities (see Section 6.2).

c) Make applications more robust to attack by distributing and replicating content (see Section 5.4).

d) Manage identity and support cross-application reputation management, non-repudiation of identity, etc. (see Section 4.1).

e) Control the extent to which each party's identity is revealed to others (see Section 4.2).

f) Allow rendezvous in a world with Network Address Translators (NATs) (see Section 5.1).

g) Allow staged interaction among machines like laptops that are not always online, and hold copies of content so it is available even if the original source is unavailable.

h) Relieve users of operational problems by outsourcing part of the application (see Section 2.1).

i) Provide the opportunity to observe, monitor and even modify (e.g., filter) content, perhaps in unwelcome ways.

j) Pre-screen incoming messages to detect unwelcome senders or unwelcome content (see Section 5.3).

k) Constrain what users can do, to create tiers of service that can be priced differentially.

l) Provide financial returns to the operators, which can drive the deployment of the application.

The ALS architecture is relevant to many other aspects of application design, as indicated by the section references in this list.

2.1 Helping an Application to Grow Up

Applications have different needs at different stages of their maturity. Many new applications start out either as pure end-to-end implementations (i.e., code running only on the user end nodes) or as end-to-end designs supported by some simple central server (e.g. to manage location, identity, or rendezvous.) But as an application becomes popular, different parties will see a changing set of needs. If an application comes to represent a significant component of overall traffic, ISPs will want to take steps (like caching) to reduce traffic loads. The application may be seen as a source of revenues by a number of players. Governments may be interested in observing, regulating, or constraining what is done using the application. Many of these interests will manifest as attempts to inject different players into the application's behavior by way of ALSs. So an application that is mostly end-to-end when it is young may grow up into a much more ALS-oriented architecture.

There is a parallel here with the need for "application management". When an application is young, there is usually no role for a person who is the "application manager". The users are the managers—they detect that something is broken and do their best to fix it. However, as applications mature and the base of users shifts from early adaptors to main-stream (less well trained and adventurous) users, there is an increasing need to outsource the management to professionals who can make the service more reliable and robust with less user involvement. Most individual users don't run their own POP or SMTP servers, although the architecture would let them do so.

A set of rules for an application might be: *it should work "out of the box" without any ALS support, when necessary, but it should allow ALSs and external management to be added incrementally as needed.*

Peer-to-peer (P2P) systems represent a very interesting special case in this spectrum of managed/unmanaged and end-to-end/ALS. P2P systems assume that there are no special machines designated as "servers", but that all the participants in the application play both the role of server and client. Users of the application help each other by providing resources. This pattern could represent a very interesting way for an application to start growing up—as enough users appear but before the application has attracted the attention of "professional service providers", the users can support each other, as services are needed beyond what the end points can do for themselves. At the same time, looking at P2P from a critical perspective, applications designed in this way may suffer from a form of arrested development; they may never be able to become fully mature. This is because the P2P design is so deeply embedded into the architecture that it is impossible for a service provider to offer a managed P2P service and attract users. In the drive to make sure that there is no central locus of control, many P2P systems remove the user empowerment to pick what services and servers they want to depend on.

A very interesting design challenge would be to design an application that can shift "automatically" from pure end-to-end to a peer-to-peer mode to a managed ALS mode as an application matures. These mode transitions should also be possible in the reverse direction: using managed services when they are well run and tussle-free, falling back on a peer-to-peer structure if the servers are missing or unwelcome, and at the end falling back to a pure end-to-end mode if only two machines are left running the application.

2.2 Application Identifiers in Initial Request Messages

Most applications have some sort of initial request message to establish contact with a remote service. Examples include an HTTP get request or a Telnet login. In early protocols like Telnet, the end-machine initiating the request did the translation from name to location, and then it just sent a message with an entity identifier relative to that location. If a user wanted to log into machine mit.edu as "bob", the Telnet client would look up the string "mit.edu" in the DNS, get an

IP address, and then connect to that machine. It might then send a login containing the string "bob", but the string "mit.edu" would not be sent. In this case, the receiver could not tell if the connection actually came to the right machine. Furthermore, it was impossible to insert an ALS or relay in the path from the source to the destination, because the ALS/relay did not know the final destination.

The first version of the Web had a similar problem: when a browser extracted the DNS name from the URL, the DNS name was not sent to the server. This meant that it was not practical to have multiple web servers with different names on the same physical machine. Later versions of HTTP included the whole URL, including the DNS part. This made it possible for a single machine to act as multiple servers by using the DNS portion to disambiguate the request, and it allowed a cache or proxy ALS to attempt to satisfy the user request.

In contrast, email messages have always carried the full destination email address in the header of the message. Any server along the path can look at the message and see where the email is supposed to go. In fact, a server along the way may have a different idea (hopefully a better idea) of where the mail should go. Without the full email strings in the email header, the relayed architecture of Internet email would not work.

This suggests the design rule:

- *Application messages should contain explicit, full application-level names.*

This allows an ALS to be interposed in the application-level connection. Assuming that this is what the users actually want, it is an important and useful capability.

3 Designing against Adversaries —Tussle

In a simple world, all parties to a communication would have consistent shared interests, but this is not realistic today. A very important aspect of application system design is protecting the application from its adversaries and their goals. The word "tussle" [Clark2000] has been used to capture the reality that parties with adverse interests vie to determine what actually happens in the network. Tussle pervades everything that users do. For example, a user may want to send mail privately, but the government might be observing it; a user might want to look at web pages in private, but the ISP may be gathering a profile of what the user is looking at; a user may want to receive mail, but not from spammers; a user might want to share music files, but the music industry wants to block unlawful distribution.

There are several forms of tussle, which can be cataloged as follows:

- Two (or more) users with common interests try to operate in presence of hostile or adverse third parties. This pattern captures the tussle between the desire for privacy and the government's desire to wiretap, for example.

- Two (or more) parties may want to communicate, but may have conflicting interests and want help. In this case, some third party may be involved to constraint or validate what is being sent. For example, a user may employ an incoming mail server that checks attachments for viruses, or a buyer and a seller may depend on a credit card company to provide protection from fraud. The end-points deliberately reveal (perhaps part of) what they are doing so a third party can become involved.

- One party wants to participate in an application, but not with some other party, who intrudes. The most obvious example of this is spam—users want to use email, but not to receive mail from spammers.

It is important for an application designer to consider how each of these patterns is manifested in the specific application. Following sections contain detailed examples.

4 Openness and Identity

Since the email application was designed for a totally open community, it provides an easy space for spammers to invade. Most instant messaging (IM) systems use a more restricted model: even though user names are globally managed, a potential recipient has to add the sender name to her buddy list before the message can be sent. In contrast to the open nature of email, this is a more closed system, somewhat Victorian in its nature—you cannot talk to someone until you have been introduced. It is easy to conceive of totally closed communities that don't even utilize global names. However, a system that does not use global names cannot easily merge two user communities later, even if they desire it, because of the possibility of name conflict.

There are, of course, proposals for controlling spam that don't involve filtering based on identity. But, since identity is an important part of many applications, it is worth looking a little more deeply at the issues there

4.1 Managing identity

A system that is not totally open needs some sort of identity verification or authentication. If one sender can masquerade as another, then controls on who can send are of little value.
One approach is pair-wise identity management, in which each recipient maintains the information necessary to authenticate all senders it will accept. This is reasonable where the pattern of communication is limited — the set of senders for any recipient is small. This pattern is the descendent of the era when a user would have a different password for any system he could log into (and the process of "introduction" often involved going and talking to a person to present real-world credentials in order to get a user name and password). When the number of pair-wise connections is great, having each recipient maintain the identity verification information for all senders is a burden for both the recipient and for the sender (who has to remember or maintain the necessary information for each recipient).

Another approach is to move identity verification to an ALS that is provided as part of the application. Most IM systems work this way: a participant's identity is validated by a central service before he/she can connect to other participants. Many multi-player game systems also work this way: identity is maintained in the game server, along with attributes of that identity, all of which can thus be vouched for by "the game".

The end-point of this process is *shared identity,* in which identity management is removed from the application and operated as a free-standing service available to multiple applications. Examples of this include the Microsoft Passport system and the competing Liberty Alliance Project. The use of SSL in the Web is an example of a solution with a mix of these features. The certificate hierarchy is available for use by any application, and the providers of browsers imbed a table of "valid" certificate authorities into each browser so that the user does not have to maintain this data for himself, but each web site may require a login from a user (a pair-wise solution), and only very few Web sites use personal certificates to validate the user to the server.

The tradeoffs among these schemes are well understood and much debated. The annoyance of having each user maintain multiple identities, one for each partner, must be traded off against the possible loss of privacy that comes from having a single identity that is shared across many of the different activities that a person engages in, a fear that is compounded when the identity is maintained by a large corporation whose motives may not be clear. The point here is that the designer of an application must be sensitive to this tradeoff and make a thoughtful set of design decisions. Different outcomes bias the tussle among different parties and will be more or less suited to different situations.

Of course, just having a verifiable identity does not solve the previous question of who can talk to whom. One of the advantages of building a shared identity system is that a shared identity can provide a basis for shared reputation maintenance systems, which are a subject of research today.

4.2 Hiding identity

Identity is something that one end reveals to the other as part of establishing communication. At the same time, one end may want to hide aspects of identity from the other end, revealing only that which is necessary in the context of the application. Which aspects of identity can be hidden and which must be revealed is a tussle space, of course. Many web sites will offer free services, but the “price” is that the user has to reveal personal information. Sometimes what must be revealed is a matter of law. In the US, it is illegal to send a fax without including the correct sending telephone number, and recent spam legislation requires that bulk mail have a valid return address. On the other hand, some chat groups don’t reveal the sender’s full email address to the group, so it is only possible to reply to the group, not the sender. Of course, the server that runs the group knows the sender’s full email address.

In general, passing communication through an ALS allows more control on what each party reveals to the other. One benefit of having IM messages go through a server is that the IP address of each party need not be revealed to the other. An IP address may not seem like an important part of identity, but knowing an IP address opens up that machine to various forms of exploration and harassment. Opening a direct connection necessarily reveals the IP address of each party to the other, unless some NAT service can be interposed.

Finally, any discussion of revealing and hiding identity must discuss the importance, in some applications, of specifically allowing anonymous participation. There are often very important social and economic benefits of allowing anonymous behavior, but since it is hard to police a world with no accountability and there is little accountability without identity, systems that permit anonymous operation may have to impose much more restrictive technical constraints up front, to prevent rather than punish misbehavior.

4.3 Naming Application Entities

In some applications like email, IM, and VoIP, the entities that the applications deal with are people, so the only namespace of importance is the namespace of users. For other applications, such as the Web or content sharing, the application deals with information entities, and there must be some way to name these entities as well as the users. The Internet provides a low-level naming scheme, the Domain Name System, or DNS. This is a system that allows a machine to keep the same name even when its IP address changes. But with certain exceptions (e.g., the MX mail records), the DNS is intended to name computers — physical end-points attached to the network. At the application level, it is very seldom a physical end-point that really needs to be identified. If the application hooks people together, for example via Internet telephony, instant messaging, or teleconferencing, the application needs to name people and the problem is to find their current location. If an application provides access to information, the application needs to name the information objects and the problem is to find a legitimate copy. If the copy is authoritative, it does not matter where in the network it comes from.

Some application designers have been tempted to use the existing DNS to name application entities, even though it may not be optimal. Thus, the Web embeds DNS names in URLs. This approach had the advantage that the Web could be launched without building a whole new name resolution mechanism first, which might have doomed it. It has the disadvantage that when the DNS name for a Web server changes, the served URLs change, so names for information objects may have less permanence than might be desired.

Many modern applications build their own directory to catalog their objects and use the DNS only as a way of finding the directory. Music sharing programs don't put the names of tunes into the DNS, nor do instant message (IM) systems put people in the DNS. Instead they use an application namespace that is tailored to its requirements, at the cost of more complexity in the design.

In these examples, the directory and the lookup service is designed specifically for each application, so there is no sharing of mechanism among applications beyond the DNS. The obvious next step is to design a mechanism for naming higher-level entities in an application-independent way. The Session Initiation Protocol (SIP) is one such example. It was conceived as a protocol to set up Internet phone calls, but it has been used to set up multi-player games, and other purposes besides VoIP. It would typically be used to name people. The URI working group of the IETF designed a location-independent and application-independent name space for objects [URI], and CNRI has developed the Handle System to provide persistent names for digital objects on the Internet.[1] So there are number of options for entity naming other than complete design from scratch that are open to application designers today.

One end-point of this design process would be to pull application-level naming down into the network layer. Gritter and Cheriton [Gritter2001] have proposed that individual information objects be given IP-style identifiers, and that the routing mechanisms in the Internet be used to compute a path to the closest copy when it is requested. This raises a variety of issues, from the size of the routing table to the concern about whether ISPs (and the Internet layer generally) should be given the total control over the algorithm for finding a desirable copy of an object.

5 Robustness Against Attack

If an application becomes successful and society starts to depend on it, then it will become a target of attack. An application designer should think about how an application can be robust enough to be "mission-critical".

Many of the issues about attacks will be specific to what the application "does", of course, and are outside the scope of this discussion. And issues of poor implementation and the resulting low-

---

[1] See http://www.handle.net/

level vulnerabilities should need no more than passing mention. But there are some issues that apply in common across applications.

5.1 Rendezvous in a Threatening World

One aspect of hiding one’s identity (or in fact one’s existence) is to put all of one’s machines behind a Network Address Translation (NAT) box. Machines that don’t have public IP addresses are better shielded from probing and attack. However, the use of NAT boxes raises an important question for any application designer—how can communication be initiated between machines when none of them are visible on the net? This so-called *rendezvous* problem is becoming more critical as more and more of the end-nodes on the net vanish behind NAT boxes.

Rendezvous refers to the process by which two machines that want to interact in the context of some application identify each other, make initial contact, and successfully establish communication. Today, the simple model is that one machine gets an IP address for another and sends an initial packet to that address. But as noted above, more and more machines are hidden behind NAT boxes and don’t even have a public IP address. And many users are uncomfortable revealing their IP address to others, even though they are willing to have some restricted interaction with them.

An ALS architecture can solve these problems. IM systems that use a central server can work even if all the machines are behind NAT boxes. Each machine makes an outgoing connection to the server, and then the server passes data back and fourth between the two. But it is much trickier, if indeed possible in general, to build an end-node-only application that can work when all the machines are behind a NAT box. Kazaa, for example, solves this problem by finding a participant that is not behind a NAT box and causing it to become a relay between the two hidden hosts. Rendezvous in the presence of NAT boxes is a current topic of discussion in the IETF.

The problem of session initiation, or rendezvous, may be handled by a set of common ALSs in the future, as an alternative to an application-specific server architecture.

Finally, rendezvous is a process that may need to be better protected from prying eyes. Due to the design of IP and the DNS, rendezvous uses public packet headers. The original DNS did not catalog services on a host, but just hosts themselves; services on a host are identified by “well known ports” that are statically assigned to applications. The well-known port visible in every packet reveals what application it is a part of. This allows ISPs to monitor what applications their users are using, block certain applications or deflect them to servers of their choice, and so on.

A recent extension to the DNS [RFC2782] allows the registration of a service on a host, not just a host; looking up a service name returns a port as well as an IP address. This could be exploited to modify the balance of power in the tussle world, by returning a random port number unique to the host rather than a universal well-known port. If the Internet worked this way, the balance of attack and defense around ports would totally change. A firewall could no longer easily block or allow specific applications for all hosts by blocking port numbers. On the other hand, port scans would be must harder to exploit, since finding an active port would give no hint as to what application was behind it. Since server ports could be picked from the full 16-bit port space, port scans would be very inefficient, in any case. ISPs could no longer track or block application usage. This might make network engineering much harder, since one could not track the growth of specific applications. On the other hand, ISPs might be more motivated to offer different QoS as part of service stratification, as opposed to blocking or enabling different applications. Finally, it would be straightforward for several machines behind a NAT box to offer the same service, since they would not have to “share” the same well-known port.

5.2 Data Visibility

There was little encryption of data in the early days of the Internet. This raised the risk of unwelcome disclosure but made “benign peeking” possible, for debugging, delegation of function to servers inside the network, and so on. With increasing concerns about disclosure, some applications have incorporated application-specific mechanisms like SSH to provide encryption under specific circumstances. And application-independent approaches like SSL are quite popular today, even if encryption is not the norm in the Web.

Some application designers have concluded that the decisions about the need for disclosure and integrity controls are not a part of the application, but a consequence of the relationship between the parties, the sensitivity of the information, and so on. In this view, the decision to use encryption should be handed off to a lower-level mechanism such as IPSec or VPN tunnels. This may be satisfactory in cases where both parties have compatible and easy means to activate such tool, but it may fail on grounds of complexity and difficulty of use.

Doing encryption right, including key distribution and protection, is a specialty that many application designers are not trained for. It is good that encryption has reached a stage of maturity that makes it available as a stand-alone system that application designers can incorporate. However, it is not clear to what extent encryption can always be extracted from the application. If an application wants to encrypt some but not all of a communication, for example, the lower-layer schemes like IPsec and SSL will not be useful.

An important tussle is the relationship between hidden content and content filtering. Encrypting the contents of messages limits the filtering that a third party can do. Among trusting parties, this limitation may be exactly what is desired—for example to sidestep government censorship. But among parties that don’t trust each other totally, filtering may be desirable. Users may want to have their email scanned for viruses, and adults may want to filter their children’s communication to prevent the receipt of objectionable material. Since the wishes of the different parties to the communication may differ, it is complex to sort out how the application should actually function in these cases. It is a question of who is in charge, how that fact is expressed, how the application responds to it, and whether the resulting behavior can be seen and understood by all the parties, or whether parts of the action are hidden.

5.3 Denial of service

Denial of service (DoS) attacks and their more aggressive relative, *distributed denial of service* (DDoS) attacks, attempt to overload a machine to the point where it cannot provide service to legitimate users. Many of the attacks today have focused on lower layers, such as TCP, and attacks on the application should be considered in that context.

The nature of most attacks on TCP is not just to overload the processor, but to exhaust some limited resource, such as the storage for TCP connection state information. The goal is to trick the TCP software into expending effort to set up needless state, which is then maintained for some time to the exclusion of legitimate requests. One of the factors that make attacks at the TCP level worse is that TCP has to set up a connection descriptor on receipt of the first packet, before it knows anything about the identity of the sender or even if the sender's IP address is legitimate. Many attacks at the TCP level involve false source addresses.

One way that these problems might be solved is to let the application provide hints to the TCP layer about what source IP addresses are more or less likely to be valid. The application might keep track, for example, of what IP addresses had ever been seen before, and instruct the TCP layer to give them higher priority in times of overload. This idea might be helpful, but would require a redesign of the lower levels or the preprocessing of incoming traffic at the packet level before it even reaches the host running the application. This option is not easy to exploit today, but it is worth keeping in mind since it implies something about what information the application may want to store for later use.

If the TCP attack problems can be solved, we can expect that DOS attacks will be directed at the applications themselves. How might an application be made more resistant to these sorts of attacks? In general, the same reasoning applies—avoid a design that requires the application code to invest much effort or create much state until it is sure that it is talking to a legitimate partner. The Web actually has a tool that is helpful here, although some users view it with suspicion: cookies. Cookies allow a user to show that the other end previously knew him, and to send the credential in the first message sent. It is worth thinking about how mechanisms such as cookies can be made robust to forgery, replication and theft.

Another class of defense is the so-called "puzzle": a challenge sent back to the client that the client must solve and return before the server will take any significant action. The idea is to demand enough effort of the attacker that he runs out of resources before the server does. The application designer has an advantage here over the TCP layer. TCP must generate some sort of response upon receipt of the first packet, but the application will not receive an incoming message until the TCP connection is established, which implies that the source address is valid. Even in a DDoS attack that uses many machines to overload the target, the source IP addresses in each of the attacks must be valid, so many attacks from the same machine can be detected as such. Of course, puzzles and related mechanisms have the effect of adding round trip delays in the connection initiation, which is contrary to most design objectives.

Another theoretical means of protection is to make a sender include in the first message a micro-payment that the recipient can refund if the sender proves legitimate. This implies a framework for micro-payments that we lack today, but it is an interesting speculation.

5.4 Defense in depth

One way to an overloaded server is to permit the server to "outsource" protection to other machines. This is an example of "defense in depth", applied to the problem of denial of service. If the incoming load of requests can be pre-processed or spread out among a number of machines, a DoS attack may be diffused and rendered ineffective.

Caches and replicated content provide a form of defense-in-depth. Since the content is replicated in many places, an attack on any single machine may disrupt a subset of users but will not disable the overall service. An application designer should consider which aspects of an application need to be centralized, which can be replicated, and how preprocessing or preliminary interaction with the source can help separate attacks from legitimate requests.

Of course, if there is a central server in the application design, it can still be vulnerable to a direct attack if the attacker can bypass the first line of defense. If the IP address of the server is known, then only low-level restrictions on routing can prevent a potential attacker from sending directly to the server. At this point, the options becomes more complex: either the server only accepts connections that come indirectly from one of the pre-processors, or routing is restricted so that all packets have to pass through some checkpoint where legitimate packets can be distinguished in some way. There are proposals to create a packet-level mechanism that works this way, for example I3 [Stoica2002]. But for the moment, the application designer is on his own.

5.5 Attack amplification

One vulnerability to avoid if possible is a design in which one node in the application can be subverted and turned against other nodes. In particular, it is critical to prevent an attacker from using a node as an amplifier for an attack, so that a successful attack on one node becomes a subsequent attack on many. In general, this requires an analysis of application level behavior, to detect places where incoming requests from one node turn into outgoing requests to other nodes. Mailing lists suggest the potential for amplification, as do any other sort of “one to many” patterns. Email attacks that use the address book of a victim to pass themselves on to others are a real concern today, and lead to analysis of attacks in terms of propagating epidemics. One class of protection, which may have as much to do with implementation as design, is to require that a human be in the loop for actions that seem to imply amplification. This will have a slowing effect, as well as perhaps detecting the attack. Limits on outgoing rates can also help, provided they can be set up so they do not interfere with normal operation.

6 Performance Adaptation.

An application designer must take into account that users may attempt to use the application under very different circumstances, in terms of performance and computational capability. Some of these differences may seem to be outside the scope of this document—whether the user has a small display or a large one, a keyboard or a speech input, a powerful processor or a feeble one, and so on. But some performance issues are directly related to the network—whether the user is on a fast or slow connection with high or low latency, and so on. Some of the issues above relate to topics of interest here.

6.1 Caching and pre-fetching

If there is a high-latency, slow, or congested link along the path from the source to the destination, the application may benefit if content can be pre-staged or cached near the user. Then the user experience can avoid these impairments. This approach makes less sense for applications that require real-time interaction, access to real time data or to another person; they make more sense with information that can be cached or with human communication, like email, that can be staged.

To implement a caching or pre-staging approach, several things are necessary:

- *The application must support some sort of caching or pre-fetching in its architecture.*
- *There must be a set of servers positioned to perform this function.*

- *The application must be able to figure out the right locations for these servers, and which server to use for a given request.*

The Internet email system has a full relay architecture built into it, but the design is not focused on the goal of staging the email “close to” the recipient. In fact, the POP server can be, and often is, at a great distance from the user. The email design was focused more on reliability, the management of the server, and do on. The Web offers examples that more directly address performance and locality. Many access providers have web content caches in their sites that can serve up web content without having to retrieve it from across the net. The firm Akamai has made a business of pre-positioning content near users, and dynamically computing which of the various Akamai servers is the best, in terms of performance, to provide the content.

6.2 Reformatting of content

If the user’s computer is particularly limited in processing capacity, display capability, and so on, and/or if this computer is at the end of a particularly constricted communication link, the application may be designed to permit the pre-processing or reformatting of incoming messages before they are sent over that slow link to the user. This concept is equally applicable to real time applications and to those where content can be cached or pre-staged. Examples of preprocessing include reducing the resolution of a video image, mixing several audio streams to produce a single one, reformatting a web page for a small display, and stripping attachments off of email and truncating long messages.

If the set of reformatting operations has been standardized, they can be implemented at the source for the benefit of the destination. But since new devices are introduced from time to time, there is a benefit of allowing new sorts of conversions to be implemented by placing them in new servers designed for the purpose. This approach, in the extreme, allows an application to extend beyond the Internet itself, to devices and end-nodes on different sorts of networks. In this case the relay point reformats the whole interaction, not just the data.

7 Economics

The common economic model of the Internet is a simple one: the user pays a single access fee (e.g. a monthly flat rate) and expects in return to get access to all the applications and services of the network. The user does not expect to pay extra for email, Web access, and so on. This is not totally true, of course; users understand that they have to pay for things like Web hosting, some third-party email services, and some content. But access to most applications comes “for free” once the user signs up for the Internet.

This bundled approach makes a lot of sense as a pricing strategy in the market—the consumer might not tolerate a plethora of small bills for specific services. On the other hand, it raises a critical question: if parts of an application are running on ALSs, as opposed to simply running on the end-nodes, who is providing those services, and what is their compensation?

The DNS is seen as so important that all ISPs just support it as part of their core business. Similarly, email is seen as so central to the net that essentially all ISPs provide POP and SMTP servers as a basic cost of doing business. But beyond that, there is little clarity or consistency in the economic structure of applications. ISPs appear to provide Web proxies both because they improve the user experience and because they save the ISP cost by reducing traffic on the trunk into the Internet. Akamai servers are paid for by the content providers, both to improve the user experience and to reduce the demand on (and thus the capacity and cost of) their primary servers.

For more recent applications, where there is no history of free use, fees are starting to emerge. Internet telephony or VoIP requires rather costly servers to transfer calls into the circuit-switched telephone systems. As a result, most VoIP services, such as NetToPhone or Vonage, have either a monthly or a per-minute fee. It will be interesting to see if SIP servers become part of the infrastructure that ISPs provide as part of their base service, or come to be a service that requires an incremental fee.

Games provide a number of examples of economic models. Some game developers sell the software and run a server as a part of the product, others give away the software but require a subscription to the service, and others use an informal scheme in which some of the players become temporary servers, thus removing the need for the game provider to provide any servers to support the game.

In this context, the emergence of the peer-to-peer systems is significant. P2P systems are based on a different economic model, which is a barter system. Nobody pays anyone else in money; each party just contributes part of the service. This is actually a very good system in some cases, although its use for the propagation of copyrighted content has given it a bad name.

7.1 Design for making money

Most Internet applications do not have a built-in model of cost recovery, commercialization, or profit. A catalog of the economic motivations that drive application providers on the network reveals a variety of approaches to making money.

- Many Web sites offer free content in order to sell advertising.
- Some proprietary applications give the “viewer” away in order to generate sales for servers.
- Some sites charge direct subscription fees or sell individual units of content for a fee (including iTunes and the New York Times archives.)

It is not unreasonable for service and content providers to make money, notwithstanding the Internet history of “free” applications. This reality begs an important question—should application design take into account the economics of the service, or is the business/economic context something that can just be wrapped around the protocols after they are done?

Building a complete model of pricing and billing into an application’s architecture is almost certainly a very bad idea. First, there will always be cases where the application is offered for free—for example inside a corporation to its employees. Second, marketing departments will always want to invent new sorts of pricing models—fixed vs. incremental payment for service, site licenses, and so on. But are there “pricing building blocks” that might be built into an application that would facilitate commercialization of applications? This topic has received very little attention, and what attention it has received has been hostile. There is a sentiment, quite possibly justified, that if it is easy to bill for an application, then providers will do so, which will erode the “bundled pricing” model that seems to work so well for the Internet.

Despite these concerns, it is not wise simply to ignore this set of issues. By doing so, designers lose the opportunity to shape how the economics of an application matures. Specifically, if there are no “economic hooks” in the protocols, service providers may try to exploit other features of the protocol to change the economic landscape, to the detriment of overall operations. Here is one example, hard to prove or document but rumored none the less. Some web hosting providers, since they are paid based on the number of hits, may be marking static web pages as “uncachable” to keep downstream providers from holding a cached copy. This phenomenon represents a tussle between two providers with different economic motivations—one wants to reduce the number of hits to the upstream copy, one wants to increase it. In this case, a correction might be to take information such as “uncachable” and imbed it into the data itself in a way that cannot be modified without detection, so that the setting represents the opinion of the creator, and not of an intermediate server. But it may take more experience than most of us have to think of these sorts of things in advance.

One specific consideration is direction of value flow. The phone system has two sorts of long-distance calls, those where the sender pays, and those, the “800” calls where the receiver pays. One of the problems with the Internet is that one cannot tell which way the value flows just by looking at the flow of packets. But knowing whether the sender or the receiver is prepared to pay for an action might help sort out a number of billing problems. Note that specifying which way value flows in no way sets the price. That can be done in a separate transaction.

8 User Empowerment—Tilting the Playing Field

The list of possible ALS functions in Section 4.2.2 reminds us that application-layer servers in the network represent a rich space in which the tussle of interests will be carried out. The application designer, by crafting the server architecture of the application, is crafting the tussle space that will emerge as the application emerges. Many network designers and application designers share a value that has been called user empowerment, or freedom of choice and action, or an open marketplace. All of these terms try to capture the idea that the Internet is a place where users can run the applications they choose and pick the providers on whom they will depend; it is not a place where monopolists gain the power to impose restrictions on user action. Of course, this is a tussle space, not a given right, but many designers see the individual user as potentially at the mercy of “big business”, or “repressive government”, and tend to make design choices that favor the end-user when they can. We need techniques to tilt the playing field toward the end-user as we design a server architecture.

For the application designer, user empowerment is the ability of the user to pick the server(s) to use and to avoid the server(s) he chooses to avoid. Implementation of that empowerment is governed by a set of simple rules concerning what is visible, hidden, and protected in messages.

- That which is encrypted (or missing) cannot be seen.
- That which is signed cannot be changed (without being detected).
- That which cannot be seen cannot be acted on.
- Aggregated content whose aggregation boundaries cannot be seen cannot be treated selectively.

Consider a simple example—a user trying to send a piece of mail. Normally, the user picks the SMTP server he wants to use. Giving that choice to the user seems right to most of us. But recently, there have been cases where an access ISP (for example, a provider of residential broadband service) prevents the user from connecting to any SMTP server but the one the ISP provides. There may be claims that this is for “reliability” or some such, but the practical reason seems to be to restrict the form of the “from” email address being used to only those provided by the ISP. By doing this, users are prevented from using a third party email service or their corporate email address, and they may be persuaded to upgrade their service to a more expensive version. One response by a crafty user is to use an SSL connection on a different port rather than raw SMTP, to prevent the ISP from seeing what is going on and blocking it. Of course, the ISP could block SSL connections, which leads to further escalation. There are many applications today that are being designed to work “over the Web”, either as an option or as the only transport mode. This may be seen simply as a demonstration of horrid design taste, or as a crafty way to commingle this application traffic with a class of traffic (http) so important that any ISP or firewall more or less has to let it through. By encrypting and aggregating traffic, the user can attempt to impose his choices of servers, despite the motivations of his providers.

All this has been worked out for email in an ad hoc manner, long after the basic email protocols were designed. But any application designer today should think through how this might work out for his application in the future.

- Should the application have a native “encrypted mode” that the sending application can easily trigger? If so, how are the keys managed? What is the relation between the key that links an end-point to a server and a key that links end-points?

- Are there parts of the message that a server may want to change (for good reasons), and other parts that ought to be signed to prevent changes?

- Should the application have a “Web mode” or other aggregated mode that commingles the traffic with that of other applications?

- How does the user “get started” in using this application? How does he find the first servers he needs? Does the access ISP provide the server (as it does the DNS), is there is a central service run by a third party (such as the application provider), or does the user manually configure his software to select a server of his choice, perhaps using information that was advertised on the Web (as happens with some email services)? Could this process of selection be automated as part of the application design?

8.1 Receiver empowerment

All the discussion to this point has been from the viewpoint of the sender. To first order, it is the sender that picks the recipient. But of course, this is too simplistic. Consider mail: the receiver picks his “incoming” mail server and configures the DNS so that the sending server connects to this server. Consider the web: queries to URLs can be deflected by the receiving end to one or another server in a number of ways. The user can be given a different URL based on where he is coming from, or the DNS that defines the URL can be programmed to give different answers based on the current circumstances. Again, most examples of this today are being retrofitted onto applications where the idea was originally missing, but a designer today ought to consider what options should be installed in the application itself.

There are also proposals to control routing at the network level to meet the needs of receivers—schemes like I3. [Stoica2002]

9 Conclusions

9.1 Application design techniques/goals

- *Try to sort out the motivations of the parties that may participate in the application.*

- *Consider the importance of user empowerment: Let the users pick who they want to communicate with and what servers they use.*

- *Use encryption as a tool to control what can be seen, what can be changed, and what is hidden. Think about the needs of trusting and wary users when making these choices. Even trusting users may benefit from an application architecture that involves servers, which may imply revealing certain parts of the communication. Consider if aggressive use of information hiding may cause the application to be banned in some contexts.*

- *Design for the life cycle of an application. Design so servers are not needed with small numbers of users, but can be added as application grows up. Consider using a peer-to-peer architecture as a stage of growth and a fallback mode, not as an immutable feature of the design.*

- *Include full application entity information in the messages themselves, rather than implying part of it implicitly by the Internet destination, so that servers can participate in the mechanism. Encryption can hide this information from prying eyes.*

- *Do not create a new form of spam in a new application. There is no reason to make it easy for unwelcome users to intrude on others, but be aware that they may have a strong economic incentive to do so.*

- *Do not provide an unwitting tool for attack amplification.*

- *Include a concept of identity, and consider giving the receiving user control over how restrictive he wants to be in what he accepts. Sometimes, anonymous communication is socially or economically valuable.*

9.2 Some Useful Common Services

Most of the common services listed here have a strong social component, and will be the target of tussle and debate. We do not mean to minimize any of that, but to invite the debate in an orderly and coherent manner by calling for the services.

- A service to manage identity and support cross-application reputation management, non-repudiation of identity etc.

- Naming services for different sorts of entities, such as SIP, URIs and the Handle System.

- RFC2872 service lookup to avoid well-known ports.

- Encryption layers that provide different sorts of information hiding and signing.

9.3 Possible Changes to the Network Architecture

Provide:

- New modes of routing to support applications.
- Routing to an identity that is not an IP address, to protect the address and control the traffic to the destination.
- A means to link application-level identity to packet-level identity, so that packet-level filtering and routing can be controlled by receiver specification.
- A service that provides abstracted information about network topology and performance, to assist application with locating servers within the application.