Internet 0

by

Raffi Chant Krikorian

S.B. in EECS, Massachusetts Institute of Technology, 2000
M.Eng. in EECS, Massachusetts Institute of Technology, 2002

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning
in partial fulfillment of the requirements for the degree of

Master of Science in Media Arts and Sciences

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
[September 2004]
August 2004

Author .....
.....
Program in Media Arts and Sciences,
School of Architecture and Planning
August 6, 2004

Certified by .....
Neil Gershenfeld
Associate Professor of Media Arts and Sciences
Thesis Supervisor

Accepted by .....
Andrew Lippman
Chairman, Department Committee on Graduate Students

Internet 0

by

Raffi Chant Krikorian

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning
on August 6, 2004, in partial fulfillment of the
requirements for the degree of
Master of Science in Media Arts and Sciences

Abstract

The Internet is currently unsuitable for small devices because the assumptions behind its architecture for desktop and server computers do not scale down. Implementations and costs that apply to larger machines have a baseline that is still too high for small embedded objects that cost only a few dollars, thereby leaving many devices disenfranchised and in disconnected groups. Similar to computer network evolution where the Internet was used to bridge together heterogeneous computer systems, I present Internet 0 (I0) as a framework to bridge together heterogeneous devices via Internet Protocols – therefore in a manner that is compatible with designing globally large computer networks.

I examine the seven following themes of Internet 0. No one of these themes are new, however it is unique to have them all present in one system:

1. bringing the Internet Protocol all the way to the device level to make devices full network citizens;
2. compiling standards and delayering network protocol stacks to make them computationally efficient enough to fit into embedded microprocessors;
3. allowing devices to talk to each other directly to remove the necessity of centralized servers and protocol converters;
4. advertising a device not only to the network world but also to the physical one to allow direct interactions between objects and objects and also objects and people;
5. slowing down networks to decrease network complexity and therefore simplify network access;
6. using the same modulation scheme across many different media so device designers can be free to choose their preferred hardware medium but not be isolated from devices that use another; and
7. pushing the engineering politics of open standards to inspire competition not in differing architectures, but in differing network services.

During this examination, I contrast these themes with the current methodologies used for device networking and propose new I0 based architectures that can replace the original solutions.

Thesis Supervisor: Neil Gershenfeld
Title: Associate Professor of Media Arts and Sciences

Internet 0

by

Raffi Chant Krikorian

Submitted to the Media Arts and Sciences department
in partial fulfillment of the requirements for the degree of
Master of Science in Media Arts and Sciences
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
August 2004

Thesis Advisor .....

Neil Gershenfeld
Associate Professor of Media Arts and Sciences
MIT Media Laboratory
Massachusetts Institute of Technology

Thesis Reader .....

Karen Sollins
Principal Scientist
Advanced Networking Architecture
MIT CSAIL
Massachusetts Institute of Technology

Thesis Reader .....

Danny Cohen, Ph.D.
Distinguished Engineer
Sun Microsystems

Acknowledgments

This thesis (and my sanity) would not be possible without a support network of people.

First, and foremost, I would like to thank Neil Gershenfeld – an excellent advisor. Without his guidance and push over the last two years, I would never have made it this far. I would also like to thank my two readers: Karen Sollins for always trying to keep me on the right academic track whenever I was bound to stray from the course, and Danny Cohen for providing me with so many thoughts that my head hurts. Both of you have been wonderful.

I owe an incredible thank you to the three UROPs I have worked with closely turning this tenure in PHM. Josh Wilson for hanging out, Jon Santiago for making me home my skills in teaching, and I cannot say "thank you" enough to Michael Tang. You have been invaluable.

Many thanks to Manu Prakash with whom I'm on the same track at PHM. Thanks for putting up with me as I berate you with administrative questions. And, of course a large thanks to the rest of the PHM: Ben, Jason, Rehmi, Yael, Ben, Xu, and Amy.

Thank you to Linda Peterson and Pat Solakoff for hand holding hopeless cases such as me. And a *huge* thanks to Susan Murphy-Bottari and Mike Houlihan for making this entire system run as well as it does.

Hugs to Shannon Cheng, Jessica Fox, Ethan O'Connor, Winter and Wyzoming for providing me with a house that I actually wanted to come home to every night.

And of course, thanks to Mommy, Babby, Nani, and Winston for years of letting me be who I am.

Contents

List of Figures

Chapter 1

The Unshrinkable Internet

The assumptions that govern the development of Internet infrastructure do not scale to the size of embedded and smaller technologies. Conventional networks take for granted their protocols and speed, however implicit in all that is the *cost* of the network – measured not only in the direct monetary cost of the hardware needed to set up the network, but also in the indirect cost of the desktop computers, of the licensing fees for the operating systems, of the system administrators needed to keep the network running, and the network installers who ran all the cabling and setup the access points. If embedded devices need to invoke all this administrative overhead, then they would be doomed to failure.

The current status quo of device network design (Bluetooth, IrDA, LonWorks, BACNet, etc.) involves the creation of proprietary networking standards and protocols either as an engineering decision or as a eco-political move to gather more market share. This has the unfortunate side effect of causing the creation of incompatible and disconnected device networks; it is currently not possible to take one embedded object and have any hope at having it communicate with another. Internet 0 (I0) aims to rectify that by introducing a framework within which to design a device network that has the same characteristics as other currently deployed device networks, but also remains open to future development while being compatible with global networking.

Before discussing the I0 framework in more detail, the design principles of the Internet should be considered first – notably the use of the Internet Protocol and "end-to-end" system design. The Internet is the inspiration for this work, and many of the main themes of this work are born out of direct consideration of the Internet's design.

1.1 Design principles of the Internet

The Internet is, by definition, the largest computer network ever built. It has two lessons to thank for its success: the use of the Internet Protocol (IP) as its communications substrate, and for guidance of the end-to-

end argument [SRC84]. IP created a uniform communications medium that could bring together the original heterogeneous networks of pre-Internet. The end-to-end argument has, for the last twenty years, provided architects and engineers a framework within which to think about networking design in a way that has allowed the Internet to grow so quickly.

IP is the "bottleneck" of networking. There are many different hardware networking implementations and there are also a multitude of networked applications, however they all utilize the Internet Protocol as their transport. This allows for both modularity and interoperability of all these networked devices – it means that no matter how a device is connected to the Internet, a computer can communicate with another computer on the network without knowing anything about the underlying network technology. All that is required of a computer to be able to join an IP network is the ability to prepend data with the extra 20 bytes of the IP header. All that is required of a networking technology is the ability to carry these particular packets. Hosts can then be interconnected by a myriad of these networking technologies to create the Internet.

Embodied in IP is also the principle of "end-to-end" [SRC84] which states that "...functions placed at low levels of a system may be redundant or of little value when compared with the cost of providing them at that low level." Saltzer, Reed, and Clark are stating that whenever you are dealing with a network of intermediaries (not only computer networks), that those in the middle should not attempt to replicate the functions which embody the intelligence of the communication as that leads a system designer down a slippery slope of replication of functions throughout this chain. Instead, a system designer should carefully construct a network where those in the center do the minimum necessary (provide transport, connectivity, storage, etc.) to allow those actually actively participating in a transaction to handle the details. Those at either end know exactly what is needed, whereas those in the center can only make assumptions and guesses – which may be found to be incorrect.

This design framework is seen in the network design of the Internet. Its classic example is the separation of the transmission control protocol (TCP) from IP by Danny Cohen. In the original specification, IP had transmission control, meaning that a packet was guaranteed to reach its destination in the same order that the source sent it (with respect to other packets from the same source to the same destination). Cohen argued that this was not useful for all applications and it was especially detrimental to real-time voice applications – instead, TCP should be be negotiated and used by the end points; the ends of the network know exactly what they need, and they should implement it whereas the center of the network simply shuttles packets around.

1.2 Scaling those principles down

Scaling the use of the Internet Protocol and the end-to-end argument was once thought to be quite difficult. IP is not used in many device networks today, perhaps demonstrating a fear of the complexity of the IP stack (as most associate an IP stack with an operating system which they do not want in their embedded devices if not necessary) and also of the complexity of an IP network. Using an IP based network invokes questions

surrounding the distribution of IP addresses and the peculuarities of network configuration. Questions about hardware and software deployment, scalability, and manageability are also very common – and all of this may seem daunting when attempting to IP-enable a light switch.

In this thesis, I present a framework to address these issues. This work differs from current networking technologies such as X10, LonWorks, CEBus, BACnet, ZigBee, Bluetooth, USB, IRDA, HomePlug, SPI, I^2C as none of them are based firmly around the lessons of the Internet and all of them will eventually face the same scaling issues in global naming, routing, and interoperability that the Internet has already faced. Internet 0 is an attempt at allowing devices to communicate at the IP level to allow them to reap the benefits of a very mature networking standard.

Chapter 2 discusses the very notion of bringing the Internet Protocol to the leaf node. I will review the IP, UDP, and TCP layers and present a method for implementing it in a simple enough fashion that it can be embedded in devices with very little computational power and memory. Compressing these protocol stacks are performed with lessons learned from Chapter 3. In the same way that compilers generate low-level machine code from a high level language, we create a small IP stack by optimizing and eliminating portions of the IP specification.

Once every device is able to speak native Internet Protocols, the network itself still has to be made very simple to install and maintain. Doing that means removing centralized dependencie – the focus of Chapter 4. Part of that is the removal of any centralized control; all interaction needs to occur with the peers themselves and not with a server in the middle of the network. To make this happen, I am introducing, in Chapter 5, a paradigm of physical identification and access to happen alongside network identification and access.

Finally, none of this would be possible if conventional networking hardware technology is required as the majority of that equipment is well above the monetary cost threshold of any given device that we wish to network. Chapter 6 proposes a device network that is "fast enough" for devices, but also slow enough to make the network simple. As a modulation scheme, Chapter 7 discusses using an ultra-wide band-like modulation scheme that can be used in an end-to-end fashion on the network.

Chapter 2

Internet Protocol to the Leaves

Running the Internet Protocol all the way to the devices is the main drive behind interdevice-internetworking. Specifying a standard protocol for all devices to talk means that all devices can inter-operate as well as operate as part of the larger global network. Historically, engineers have stayed away from using native IP in all portions of a device network or system because of the fear that implementing the stack was too difficult to do properly[1], however this mentality has unfortunately left networks fragmented in a myriad of different protocols – almost resembling the days of networking before the dominance of the Internet. No two device networks can talk to each other, without a stateful protocol converter sitting between them and whenever ether side changes its protocol it requires that the converter be upgraded also. That converter becomes the bottleneck to the entire system. The Internet Protocol is meant to fix that by introducing a single generic data format that can be used across all networks.

While fears of implementation complexity may have been true in the past, it has been possible to create an IP stack (with UDP and/or TCP parsing) that can fit on a $2 8-pin microcontroller for a while now[2] and following Moore's law will only provide cheaper and smaller microprocessors from here on out. This means that it is inconceivable to *not* consider using IP. If we wish to be using the Internet Protocol all the way to the leaves of any network, a simple and cost-effective solution must be provided so that any relatively cheap processor can natively parse and react to these packets.

2.1 Why not IP?

Many dissenters believe that IP should not be used because the IP header alone is 20 bytes (see table 2.1) that must be prepended to any data along with an additional 12 bytes or 20 bytes for UDP (table 2.2) or TCP headers (table 2.3). If the data being transmitted is smaller or possibly not significantly bigger than that byte

[1] IP is almost always used now to the leaf node of desktop and server computer networks, it is not however, always deployed to the leaf nodes in networks of embedded devices.

[2] John Romkey connected his toaster directly to the Internet in 1990.

limit, they argue, then it is possible that transmitting IP is not the proper thing to do. Especially if operating on power-constrained links, they believe that it may be undesirable to transmit the extra 20 to 40 bytes as there may be a hard and fast power budget that this would violate.

2.1.1 IP compression

Compression of IP headers is possible in a standard way [DNP99], however the compression received is only meant for long-lived TCP/IP connections as it recognizes that in a long-lived flow values in the header are rarely changing – it simply sends the changing values of the TCP/IP header, which optimistically can be reduced down to 5 bytes per packet. For these reasons, it does not function as well for short lived TCP/IP flows, nor for UDP connections[3]. For even tighter compression it is possible to use an entirely compressed stream thereby compressing not just the packet headers, but the entire data also.

The side effect of any type of compression is the requirement that either side utilize more processing power and also more memory. Not only will one side need to compress and the other side need to decompress, but both sides will need to keep very careful track of the state of the system in order to properly compress and decompress the data.

2.1.2 Transmitted bits $\neq$ power

Secondarily, it should be recognized that the transmission of the bits is not the most power consuming portion of a radio. It is actually the amplifier that is needed to *receive* the signal. There are two variables that need to be accounted for: the first is the amount of power needed to turn on the amplifier and the second is the amount of power needed to keep the amplifier running. The receiver will want to jointly optimize the amount of time that the amplifier is left on (the constant power draw) as well as the number of times the amplifier is switched on (the large transient draw).

There are also additional strategies that can be used to minimize the energy needed by the receiver. It is possible to scheduling time intervals at which point it should turn on its amplifier to listen for a packet, and that packet can also contain a hint from the transmitter as to how long the receiver should keep its amplifier on in order to receive all the data that it intends to transmit. This also reflects an observation that there may be many information sources on the network that are not necessarily information sinks. Those information only sources do not require the amplifier as part of their design. They could, instead, have a non-amplified receiver that is only used for short range communication on programming or for physical access (see chapter 5).

[3] Although one could imagine an application layer compression scheme based about RFC2507 that is used for UDP "flows".

2.2 Notes about the IPv4 header

IPv4[4] is a relatively simple protocol as all it provides are two services to the network: addressing and fragmentation. Addressing deals with the location of a "name" on the network (it is the job of higher level protocols and applications, such as the domain name system, to handle the mapping from names to addresses) while fragmentation handles the delivery of packets which are too large to traverse a given network. Anything and everything else (such as reliability, flow control, etc.), are left to higher level protocols on the network – if they are even desired and necessary.

Before delving into the implementation a micro-IP stack, it is helpful to have table 2.1 as a quick review of the contents of the stack itself. Each of these fields in the IP header serve a very particular purpose, and when one is implementing a fully functional IP stack, each field needs to be processed. However, there are many simplifying assumptions that can be made about the values in those fields when building a micro-IP stack.

2.2.1 Type of Service

This field indicates the quality of service this packet is supposed to receive and while it is possible to implement the quality of service (QoS) on the receiving end by preferentially processing incoming packets, this field is typically used by the network itself to allow certain packets to pass through routers at a prioritized rate [AGK99]; certain research implementations do use this ToS field [SD94], however it is widely ignored in standard implementations. It is quite safe to set the value of the bits to be 0 (normal and routine delay, throughput, and reliability) when transmitting an IP packet, and completely ignoring the bits upon reception of the IP header.

2.2.2 Identification and Fragmentation

Fragmentation deals with the situation where IP is being used to bridge together two networks that have different maximum transmission units (MTUs). For example, if a network that allows maximum packet sizes of MTU_1 bytes is connected, and is required to send packets through a network that only allows MTU_2, where $\mathrm{MTU}_1 > \mathrm{MTU}_2$, it is possible that some packets traversing that boundary are too big and the gateway may need to fragment certain packets. As defined by IPv4, the gateway is required to divide the d bytes of data in the IP packet under the following conditions

$$p + \text{size(IP header)} \leq \mathrm{MTU}_2$$

$$np \geq d$$

[4]While this work was tailored to the fourth version of the Internet Protocol, it is relatively straight forward to extend it to the sixth version. That does remain as future work.

Table 2.1: The various fields, and their associated sizes, in the IP header.

Header field	**Field size**	**Field contents**
Version	4 bits	this represents the version of the IP header that is being sent. In the case of IPv4, these four bits are always set to 0b0100 (4).
Internet Header Length	4 bits	the length of this particular IP header in 4-byte words. It is almost always safe to assume that these four bits will be 0b0101 (5), meaning the header is 20 bytes long. This value will only be different if there are IP options present in the IP header.
Type of Service	1 byte	a representation of the quality of service that this packet desires. A packet can ask for variances on the amount of delay, throughput, and reliability that it receives while being relayed through the network.
Length	2 bytes	the length of this entire packet in bytes.
Identification	2 bytes	a tag that can be used to reassemble fragmented packets; if a packet has been fragmented, then this value can be used to determine which fragmented packets belong to the original packet as the ID will be the same across the fragmented versions of an original packet.
Fragmentation Flags	3 bits	bits stating whether this packet has been fragmented, or whether this packet will allow itself to be fragmented. If a packet does not allow itself to be fragmented, and it is necessary for it to be so, then the intermediary that needs to break up the packet will simply drop it.
Fragmentation Offset	13 bits	these bits represent the location of fragmented data (if applicable) within the original unfragmented data buffer.
Time to Live	1 byte	the maximum number of hops that a packet can be transmitted through on the Internet to prevent looping and lost packets. This value is decremented by one every time this packet is routed through an intermediary host, and if this value ever reaches 0, then this packet is dropped.
Protocol	1 byte	this details the protocol that this IP packet is wrapping; this byte could say that the packet is wrapping a packet of UDP, TCP, or any other protocol.
Header Checksum	2 bytes	here is contained a 2-byte one's complement of all the bytes in the IP header. These two bytes are used to make sure that the packet header has not been damaged during transmission.
Source Address	4 bytes	the IPv4 address of this packet's sender.
Destination Address	4 bytes	the IPv4 address of this packet's destination.

$$\frac{p}{8} \in \mathbb{N}$$

The first $n-1$ packets will have exactly p bytes of data in them, and the nth packet has the remainder $(d - p(n-1)$ bytes). Each of the new n packets all have the identical IP header as the original packet, except that

1. the more fragments bit must be set in the first $n-1$ packets (not in the nth packet),
2. the fragmentation offset field in all n packets must be set to pw where w is the number packet ($w \leq n$), and
3. the checksums must reflect these changes.

The IP sender (unless it is the bridge – which this micro-IP stack is not meant for) never needs to concern itself with fragmentation rules. It merely needs to transmit packets which are smaller than the MTU of its attached medium. The IP receiver, on the other hand, does need to concern itself with the above, as it is it's job to reconstruct those fragmented packets into the larger packet before handing it off to a higher level.

Defragmentation can be quite memory consuming as the receiver needs to reassemble all packets with the same identification field into a single large packet. It may be possible, if the packets are received in the proper ordering (as dictated by the fragmentation offset field monotonically increasing) to process each packet individually – however if the packet IDs are not increasing, then it will require out of layer knowledge as to what to do with the packet. In the worst case, the received will need to buffer any out of order fragments and wait until the in order fragment appears. For this reason, some micro-IP implementations may choose not to implement IP defragmentation, and may simply ignore any packet which has more fragments bit set, or has the fragmentation offset field set to a non-zero number.

The transmitter, on the other hand, should use a monotonically increasing value in the identification field for each packet it plans to send (this is easily accomplished by having 16-bits of state initialized to a random value at start-up) in case fragmentation does occur down the line. As a micro-IP stack is usually tailored to sending and receiving small IP packets, this stack should also set the "don't fragment" bit in the control flags field.

2.2.3 Time to Live

The time to live (TTL) field, is yet another one of those fields that the sender or receiver of IP packets need not worry about (unless it is an intermediary host). This field was put in place to prevent packets from getting lost and retransmitted in a loop on the network. When passing an IP packet, a router (or any other host) needs to decrement this TTL field and then update the checksum on the packet accordingly. If the TTL field reaches 0, the packet is dropped and the host (optionally) sends an ICMP packet back to the sender to inform it that it has dropped the packet. As long as the sender sets the TTL value "high enough", then the packet should be

delivered to its destination[5]. The receiver never needs to concern itself with the TTL field, as an intermediate host would have dropped the packet if the TTL was not set high enough.

2.2.4 Checksum

The checksum field are two bytes that contain the one's complement sum of all 16 bit words in the packet's header. When a packet is received, the receiver must confirm that the checksum is correct *and* that the packet is addressed to it. Neither one of these on its own is sufficient as a packet could be addressed to this node through an error in the packet header contents.

Again, for memory reasons, the checksum may have to be computed on the fly as the packet is being received. As data is streaming in, the processor may compute the one's complement sum on two bytes at a time for the first 20 bytes (or more, if the IP header is longer). If at the reception of those first 20 bytes, the sum is not 0xFFFF, the packet can be immediately discarded.

In the transmission of the packet, the checksum will need to be computed before the transmission of the entire header; the checksum cannot be computed on the fly, as the checksum is embedded in the header itself and does not trail it. However, all the data necessary to compute the checksum is known at the beginning of the transmission, so the computation of it only requires ten 16-bit additions with carry.

2.2.5 Addresses

The IP address gives a clue to the network locality of a particular node. In the current use of IPv4 (along with classless inter-domain routing [FLYV93]), the address is denoted by four bytes which correspond to which portion of the Internet hierarchy that address is connected to. Routing is not the concern of the sender or the receiver as it either places packets on the ether, or has packets delivered to it by the ether. All that the sender or the receiver needs to concern itself with is addressing a packet properly, or whether an incoming packet is addressed to it.

Determining whether the packet is addressed to it requires the receiver to determine if the four bytes matches the receiver's IP address, or if the packet's destination matches the broadcast address of the subnet that the node is on. For example, if a node has an IP address of $a.b.c.d$ on a subnet with a network mask of $w.x.y.z$, then it can either receive packets that are addressed to $a.b.c.d$, or to the broadcast address of $(\sim w|a).(\sim x|b).(\sim y|c).(\sim z|d)$ (replace all positions with the "0" bits in the netmask with "1" bits in the address).

2.3 Implementing a micro-IP stack

Given the notes from section 2.2, it is straight forward to construct the state diagram of the receiving side of an IP-stack as shown in figure 2-1. There are a few assumptions that are made in this diagram:

[5]Linux is quite conservative as its default TTL value is 64, where as Microsoft Windows sets its TTL values in the 128 range.

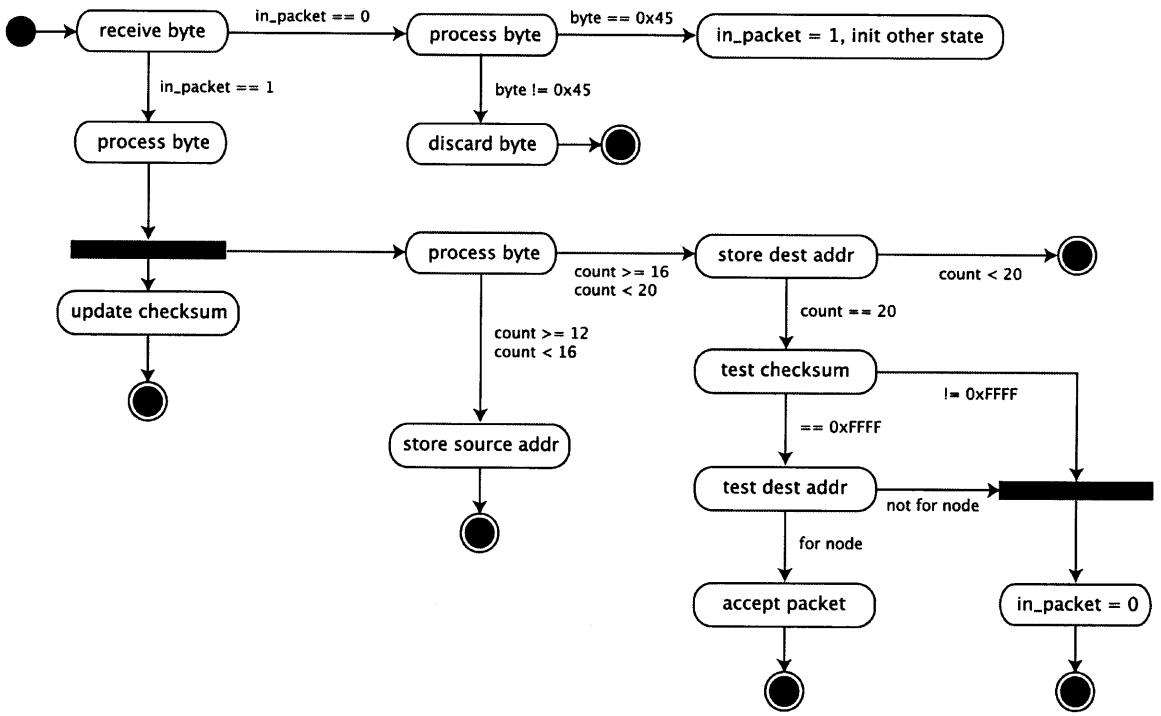


Figure 2-1: The state diagram of the receiving end of a micro-IP stack

1. An "out of band" (with respect to the IP packet) mechanism is used to signal, at least, the beginning of the IP packet such as the ones found in SLIP and Ethernet framing;

2. packets are delivered one byte at a time;

3. and fragmented packets are ignored.

Transmitting packets from the IP stack does not justify a state diagram out of simplicity. Before transmitting, the node needs to know the protocol that the data is encoding (TCP, UDP, etc.) along with its corresponding protocol value (0x06 for TCP or 0x10 for UDP) [RP94], the destination address for this packet, the length of the data to be sent, and the knowledge that the data can be streamed out upon request (the data has to be streamed out as a requirement of this stack is that it cannot buffer the entire packet to be transmitted in memory). The checksum can be partially precomputed as most values are constant – the only values which may change are the length of the packet, the packet identifier, the protocol, and the destination address – those values can be used to update the precomputed checksum easily. Therefore, the IP stack need only start transmitting a series of bytes: 0x45 for the version and Internet header length fields, the two bytes for the length of the of the data, the current two bytes for identification that the processor is using, 0x00 for the flags and fragment offset, 0x3C (or something similar) for the TTL value, the appropriate protocol value, the computed checksum value, this node's IP address, and the destination IP address. Once all this has been sent, the transmitting data can be simply streamed out after it.

2.4 TCP vs. UDP

Almost all packets found on the Internet are either user-datagram protocol (UDP) [Pos80b] packets or transmission control protocol (TCP) [Pos81d] packets. The former provides a simple wrapping around the IP protocol along with the added abstraction layer of ports on top of addresses, and the optional protection against corrupted data through the use of the UDP checksum. TCP, on the other hand, provides a reliable "connection" – the TCP stack allows the sender and receiver to communicate via a stream that then gets packetized and delivered only to be reassembled exactly as it is sent with the property that the bandwidth can be allocated between other TCP streams simultaneously. The TCP stack guarantees that packets will be processed by the receiver in the same order as the sender transmitted them and without error in the data.

However, there are trade offs between the two. UDP is connectionless, meaning that any associations between packets has to be handled by the application layer. There are no guarantees that the packet will ever get delivered to the destination host, and there is no mechanism for the transmitting host to know whether the packet arrived. Additionally, the UDP checksum may be explicitly set to 0, which means that the packet is transmitted without any checksum computation, and therefore the data may be corrupted.

TCP allows the sender to write a stream of data, and the stack handles the division of the data into packets and transmits them to be reassembled on the far end. To achieve this reliable transmission a connection first needs to be setup between two hosts (which involves the transmission of at least 2 packets back and forth – the SYN and the SYN/ACK packet) before any data can be transmitted. Then as data is being sent, the receiver needs to acknowledge using ACK packets. However, this expense is well paid when attempting to establish a SSH connection, or when moving large amounts of data as in both cases it would be horrendous if data was lost or corrupted in transit.

While all the error recovery and suppression in TCP is very useful, it does make designing applications that require real-time updates to be nearly impossible. Consider an example application which monitors the price of a stock on the market. It is possible to transmit the second-to-second varying stock price over a TCP connection, however once one packet gets lost, all other packets are held up until the sender and receiver can recover from that particular error; the sender and receiver are attempting to retransmit data that, by the time that it is recovered, will be stale. If, on the other hand, the real-time data is sent using UDP packets, each packet will need to have a sequence ID (or time stamp) associated with it so the receiver can know when one packet is newer or older than another (as the ordering of the delivery is not guaranteed), however if one of the packets gets lost, the receiver need not worry – it simply waits for a newer packet to come along. The same argument applies to real-time audio and video transmission over a network. If using a TCP stream, when one portion of the media gets lost, then the rest of the stream is held up causing this stream to be no longer in real-time. On the other hand, if UDP packets are used, and if a packet is lost, then the receiving application can simply insert a burst of static – the person or people listening and watching will notice a slight disturbance, but then the stream will progress "live".

There are cases, however, where making the decision is not so straight forward. Take the common Internet

Table 2.2: The various fields, and their associated sizes, in the UDP header.

Header field	Field size	Field contents
Source Port	2 bytes	the port which sourced this packet. This field is optional, and can be left as 0.
Destination Port	2 bytes	the port to which this port is addressed to.
Length	2 bytes	the length, in bytes, of the UDP header and associated data.
Checksum	2 bytes	the 16-bit one's complement of the UDP pseudo-header and data.

0 example of a light-bulb and a switch. On an error-less and congestion free network, transmitting a UDP packet from the switch to the light to ask it to turn on seems appropriate as it is small and not complex. However, when this switch gets installed in a commercial building with many simultaneous events occurring on the network, there is a chance that the packet may get lost. It may prove to be undesirable for a user to attempt to turn on or off the lights, but then have nothing happen when the packet gets lost. What this boils down to is where should the acknowledgment and retransmission occur – should it happen at a TCP/IP level, or should it happen at "layer 8"[6]? At layer 8, the person will notice that the lights did not turn on, and have to press the button again. The status quo between lights and switches may be maintained if TCP/IP is used between the light and switch, at the cost of a significant increase in network traffic. Only one UDP packet is needed to turn the lights on, where as at least 5 packets are needed if done via TCP[7]. Needless to say, this is a debate that is not to be settled easily – therefore, to remain compatible with either, both a micro-UDP and a micro-TCP stack needs to be discussed.

2.5 Implementing a micro-UDP stack

As said before, UDP only adds two items of consequence to IP – it introduces the notion of ports, and it provides an optional checksum over the data in the packet (see table 2.2). Because of this simplicity, it is fairly trivial to implement a micro-UDP stack as shown by the state diagram in figure 2-2.

Upon reception of the packet's bytes, the source and destination ports are stored[8], the length of the packet is recorded, and finally the UDP checksum must be recorded too. The length of the packet is necessary because it will tell the receiving stack how many bytes it should be reading off the network before resetting all state in the micro-IP stack. The UDP checksum comprises both a pseudo header (containing the source IPv4 address, the destination IPv4 address, the protocol, the total length of the data, the UDP header), and the UDP data. If at the end of receiving all the required bytes the checksum is not 0xFFFF, then the UDP stack must signal that all the data from the last received packet is invalid. If the UDP header reports the UDP

[6]Ben Bauer and Andrew Patrick have proposed human-factors extensions to the seven layer model [BP].

[7]SYN from light switch to light bulb, SYN/ACK from light bulb back to the switch, ACK with data to the bulb, ACK with FIN to the switch, and FIN back to the bulb again

[8]This is an application specific choice as some programs will react differently depending on what port number sent the packet. Other programs simply do not care and can safely ignore.

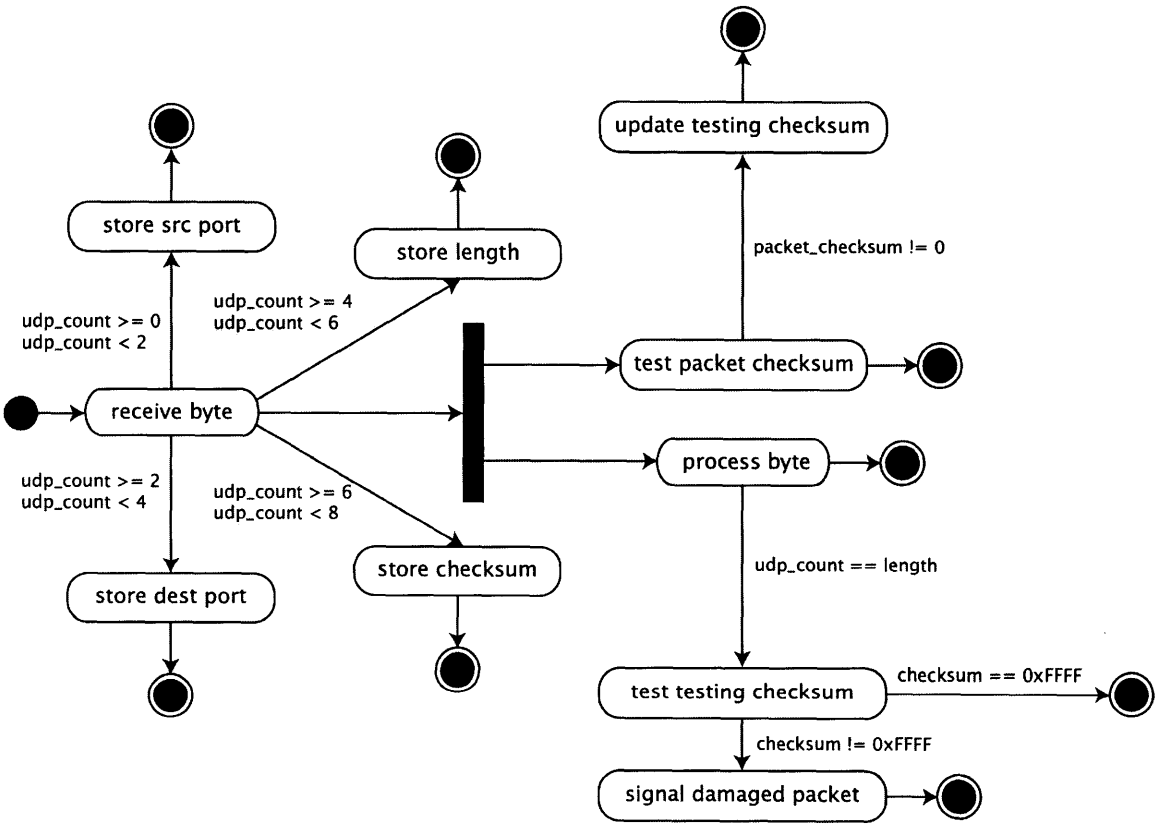


Figure 2-2: The state diagram of the receiving end of a micro-UDP stack

Table 2.3: The various fields, and their associated sizes, in the TCP header.

Header field	Field size	Field contents
Source Port	2 bytes	the port which sourced this packet.
Destination Port	2 bytes	the port to which this port is addressed to.
Sequence Number	4 bytes	the data byte number that is being transmitted.
Acknowledgment Number	4 bytes	if the ACK bit is set, this contains the number of the byte up to which the receiver is acknowledging has been received.
Data Offset	4 bits	the number of 4-byte words in the header. This is almost always set to 0b0101 (5) unless there are TCP options set in the header.
Explicit Congestion Notification	2 bits	used by an intermediary host to note that its queues are getting full.
Control Bits	6 bits	signals the state of the packet – whether it contains an ACK, SYN, etc.
Window	2 bytes	the number of data bytes that the receiver is willing to accept before it will send an ACK back.
Checksum	2 bytes	the 16-bit one's complement of the TCP pseudo-header and data.

checksum as 0, it means that no checksum computation need be performed.

Transmitting UDP packets is quite simple, with the only complication being whether the UDP checksum is to be filled in with a non-zero value. If the checksum is to be filled in, then it would be prudent to know the 16-bit one's complement sum of the data before it is transmitted so the data does not have to be buffered in memory for the computation. With that data, it is possible to compute the rest of the checksum.

2.6 Implementing a micro-TCP stack

The micro-TCP stack is a bit more difficult to implement than the micro-UDP (section 2.5) for two reasons: the amount of state necessary, and the act of synchronizing two parties over a lossy channel. Both the sender and the receiver need to keep track of the information on the address/port pair[9] so as to attempt to keep synchronized in what state they believe the connection to be in.

2.6.1 Micro-TCP receiving

The receiving end of the micro-TCP stack (as illustrated in figure 2-3) has the job of accepting the packet, acknowledging the start of a connection, making sure that we are receiving the right "number" packet, accepting the data in the packet, and also confirming that the packet has not been damaged in transmission. State has to be tracked on a connection by connection basis that includes the following

1. The source IP address of the connection,

2. the source port,

[9]Simply having a series of packets between two hosts is not enough to constitute a connection – there needs to be an association between address and port pairs so as to be able to distinguish multiple streams between two hosts.

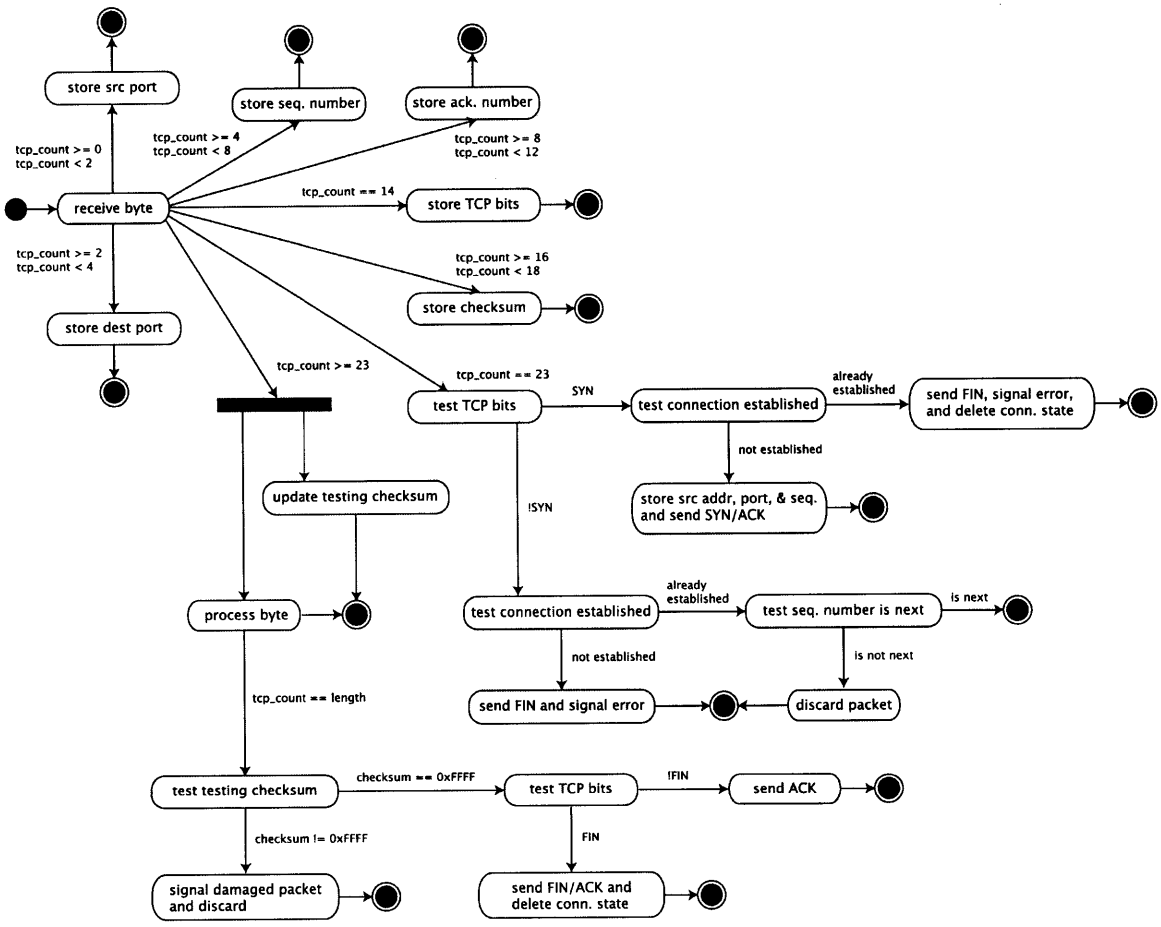


Figure 2-3: The receiving end of a micro-TCP stack.

3. the destination port,

4. the current sequence number, and

5. the state of the connection (whether it is still in the "establishing" state, "established" state, or "closing").

All the receiving state can be tracked in less than 13 bytes[10] for each connection that the stack would like to deal with simultaneously. A very simplistic stack that knows that its clients will only be sending one packet messages can get away with only attempting to manage one, or a few, simultaneous connections. But any stack that needs to deal with larger streams, will want to deal with multiple data structures.

If all the receiving state data structures are full, then there are two appropriate strategies for dealing with this connection. The first involves the creation of a "pending" queue where the source address and port of the connection is recorded in a FIFO, and if a data connection structure is freed before the TCP timeout, then an item from that queue can be dealt with. However, if an item cannot be dealt with before the timeout, or if a queue is undesirable in the first place, then the packet should be explicitly dropped and rejected back to the source with a FIN packet and the requester is free to try again at a later time. It is also recommended that a timer be also used to "time-out" connections that have not been heard from for a while as processors using the micro-TCP stack may run out of resources or suffer from particular denial of service attacks [MVS01].

For simplicity's sake, the stack keeps reporting the window size back as one to prevent multiple packets from being transmitted simultaneously without ACKs[11]. Having a window size of one has the property that when a packet is received who's sequence number does not exactly match the next sequence number that the packet can be buffered (if there is enough memory to do so), or simply silently dropped under the assumption that the sender will retransmit it when appropriate.

There are only two different failures that the stack watches for: a SYN packet over an already opened connection and receiving a packet without a SYN over a non-open connection. In both of these cases, the stack responds by sending a FIN back, and then terminates the link. The stack does not honor any other TCP options or extensions, and the connection rate will be limited by the need to keep only one window slot open and the processor's speed.

2.6.2 Micro-TCP transmitting

When a TCP/IP stack wishes to transmit data to another host, it first must start the transmission by sending a TCP SYN packet to the requested IP address and port number. From there, the state diagram of the transmitted can be represented by figure 2-4. The required state on the transmitter mimics the state needed by the receiver as both need to keep track of the same information.

[10] 4 bytes for the source IP address, 2 bytes for the source port, 2 bytes for the destination port, 4 bytes for the current sequence number, and 2 bits for the connection state.

[11] Having a larger window size does mean more throughput on the data (to a limit), but it also means more state that is accrued on both the receiver and the transmitter.

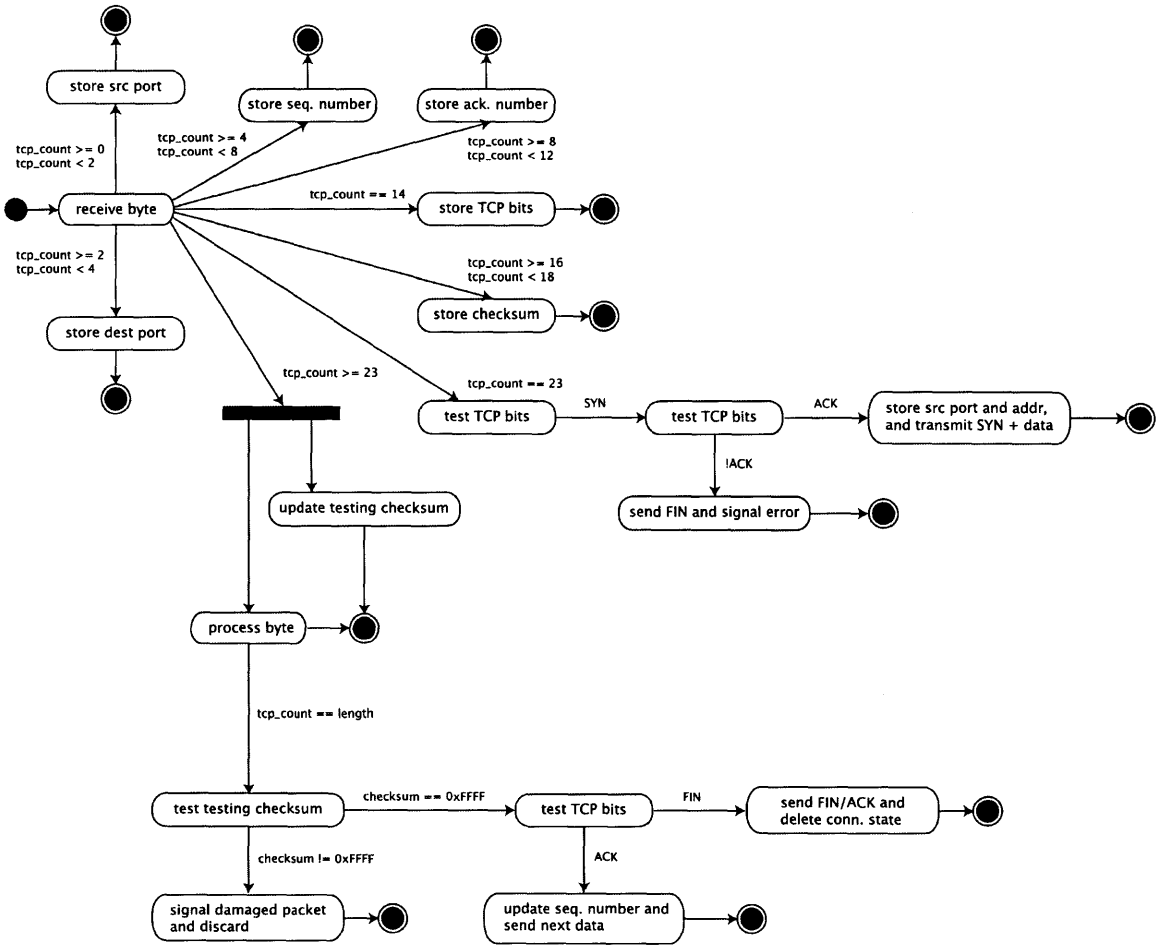


Figure 2-4: The transmitting end of a micro-TCP stack.

The complication on the transmitting end is that the receiver dictates the size of the TCP window to be used. This poses a huge problem for memory-constrained devices that cannot buffer more than a packet's worth of information for the retransmit, and the only solution here is to completely ignore the window size and only send one packet at a time and wait for the acknowledgment of that packet before proceeding any further. Unfortunately, this will greatly limit the speed of the connection – but that should not be a large issue for most Internet 0 applications.

2.6.3 Bidirectional micro-TCP

Both of the previous two sections (2.6.1 and 2.6.2) have each individually discussed either the receiving or transmitting side of the TCP stack. It is possible, however, for either end of the TCP connection to transmit and receive data – that requires a blending of the two state diagrams. As the same data structures are being used on either side of the connection, it is relatively straight forward for either side to transmit and receive. The two state diagrams can be run simultaneously on both sides, the only thing to note is that on either side, ACKs and data can both be transmitted on the same packets that are going out. It is also possible to have a half closed connection where one side of the connection issues a FIN packet to the other side. However, if the other side is not ready to terminate the connection yet, it is free to keep it open and transmitting data.

Chapter 3

Compiling standards

A compiler's job is to take instructions written in a higher level language, and produce the implementation of those specifications in a much lower level language. Along compiling the code, there are optimizations that can be run along with it – certain "tricks" that can make the code run even faster, take up less memory than specified, or even remove whole sections of the software that are not being used. Compilation is the one unifying concept to take away from the micro-IP/UDP/TCP implementations (sections 2.3, 2.5, and 2.6) as each one of those programs is an optimized and distilled version of the RFC which explains how that specification should work ([Pos81b], [Pos80b], and [Pos81d]). Each one of those implementations perform *almost* exactly as the RFC dictates, just not explicitly. At no point was the packet handled by one layer in the ISO/OSI model at a time. Instead, the packet is handled by cross-talk in the layers – information is gathered from a combination of the layers to make processing more light weight.

Most software systems are designed with abstractions, however, because that makes it simple to handle. The by-product is usually a larger compiled code size, which also requires either a bigger processor, more memory, hardware complexity, and cost. The smaller and simpler the compiled code can become, the simpler and cheaper the processor to run it can be.

3.1 Abstractions

Abstractions are necessary in order to make a project manageable by a person or by a team of engineers – without them there would be no way to build scalable modules that can be individually verified and then integrated. These separations take form in everything from the UML design documentation, to the language being used for the software engineering, to the "interfaces" and method signatures being maintained between the different developers working on a project.

3.1.1 Language choice

Language choice is one of the largest abstraction decisions one can make. Given a particular hardware architecture, one could write low-level machine code or move one level up to one's choice of a particular assembler variant – both of these is very wedded to the processor being used, and the software written is not portable to any other processor (or even other arrangements of processor and supporting hardware), however this software is bound to operate very quickly.

At a higher abstraction layer sits languages such as C which allows authors to write (most) code in a way that is independent of the architecture that is to be executing the code. The author simply takes for granted that there is a compiler that can convert those procedural statements into assembly code – however the real measure is how small can the C compiler make the assembly code. Can it make it smaller than a skilled assembly programmer? Most of the time not, because there are limits to how much as a compiler can infer about a written piece of software written in C. Languages also do go to higher levels reaching LISP and Java which are very far removed from the actual hardware. Java, for example, runs inside a virtual machine which itself is just another layer of software that is being executed on the hardware itself.

Any large software project written entirely in assembly will not be able to be scaled outside the management of a single engineer, simply because the language does not provide any facilities that aids it. It is usually very difficult for another engineer to look at an isolated block of assembly code and understand what is truly occurring in the processor as there are many non-obvious dependencies and no protection being enforced by the compiler. Those reasons and because all the vagaries of the hardware are encapsulated and hidden by the compiler[1], are the two reasons to go with compiled code.

It does need to be recognized, however, that this language abstraction layer does add some overhead as a compiler can only do so much and only the actual engineer who is writing the software can gruel know what goal he is attempting to meet by authoring the given piece of code.

3.1.2 Modularity

Another very common form of abstraction in software systems is modularity. Modularity allows a team to divide up a large system into separately develop-able pieces that can be individually written and tested before integration into a larger project. This abstraction has fans in both management and engineering because it both allows multiple people to be working simultaneously on different portions of a project (thereby speeding up development time), and it also allows software errors to be located more quickly as each piece of a system can be individually examined.

Software engineers work in a modular environment by defining "interfaces" between different code segments. One engineer way know that another is expecting that he can call a certain function with a given signature in his code – so the first engineer structures all his software around that one entry point into the

[1] For example, the Linux kernel can be cross compiled for over ten different architectures from a single code base.

software. The other engineer can then take for granted that when he calls that given function, a defined set of operations will occur. Neither of them is required to know the details of the other's operations in that the first engineer need not know anything about the calling function, and the second engineer need not know anything about the called function. They just need to agree on this interface.

While the advantages of modularity are very clear, it too also leads to larger software sizes. State can be accidentally replicated on both sides of an interface with both sides tracking a variable or set of variables when both could be sharing data. Processing time could be wasted attempting to handle an error that one side knows will never occur. And finally, features could be implemented on either side that neither side is actually using.

3.1.3 In the IP stack

The conventional ISO/OSI network model involves seven layers as show in table 3.1. Traditionally, each of these layers are specified and implemented by different parties who simply agree on the interface between the two layers. Hardware engineers make the decisions about the inter-connection of devices at layer 1 and their corresponding software engineers implement the functionality of the "driver" that is to be installed on the computer to control that piece of hardware. A third group (usually within the development team working on the operating system) implements library which handles the network layer and the transport layer. And then finally, application engineers work from there on up.

This myriad of people working within this framework can lead to a nightmare if not for the agreed upon interfaces and abstractions. Application engineers need not know anything about layers 4 and 3, only that they will perform as expected and that there is a standardized API to talk to those layers. Operating system engineers working at layers 4 and 3 should not need to know the exact details of how layer 2 is operating, only that they have defined an API that the layer 2 device driver is utilizing. And finally, layer 2 engineers may not absolutely nothing about the intricacies of hardware, but they do know about the interface that they have negotiated with the layer 1 engineers.

These abstractions and agreements allow for a very generic operating system that will allow any network application to run. The API to access the network layer provides methodology to deal with failures and reports detailed error messages if anything goes wrong (such as two different programs attempting to open the same port, etc.). A program also needs to operate only on a specific level, leaving the uncertainties of the other layers (uncertainties on how they operate or the particulars of their operation) to other parts of the operating system.

3.2 Removing abstractions

The implementation of the micro-stacks in the previous chapter show why removing these abstraction layers and "compiling" the code is such a huge win. Given a specification of the network interface and the physical

Table 3.1: The seven ISO/OSI network layers.

Layer number	Layer name	Layer purpose
1	Physical	this layer is the physical medium itself, and usually involves discussion of the connectors between the medium and the computer.
2	Data link	here the representation of the data on the medium itself is discussed. For example, all notions of Ethernet frames, and Ethernet addresses are discussed in this layer.
3	Network	IP fits in at this layer. Consider this to be the bottleneck between all the layers – there is a multitude of physical layers (which spawn a variety of data link layers) and there is a multitude of different applications that sit above IP.
4	Transport	now that all the data is coming in as packets, this is where the sockets sit and wrap all the packetized data as UDP or TCP or via any other API.
5	Session	from this point on, we have data in the network, but it has yet been undefined in what format the data is formatted in. This layer is involved in defining that information.
6	Presentation/Syntax	a common problem with computer networks is the representation of data [Coh81]. This layer's specific purpose is to make sure that all that data gets transformed in a way that the application layer (and higher) and the session layer (and lower) can both be happy with.
7	Application	finally, this is where all the applications (usually software) lives. As the final end point, this is where authentication and privacy are handled and this is where network services are handled.

interactions needed to make a light switch and bulb work, a very small and specialized code base can be created to make that happen. Analyzing a light bulb and light switch that uses HTTP GETs to intercommunicate and SLIP-like network framing, we can begin to remove items from a conventional implementation. The HTTPd can have features such as PUT and POST removed, and the string parser for the GET statement can be configured to look only for the specific URL that the HTTPd responds to (otherwise simply return a HTTP 404 error). Out of the networking layer, the UDP parser can be completely removed, all code to deal with packet window scaling can be removed from the TCP stack, the IP layer does not need to deal with IP fragmentation, and the list goes on. By breaking the abstraction layer, it is possible to determine what features and requirements are needed of the lower layers in the network, and remove those that are not necessary, and compress those that are necessary by allowing cross-talk.

Putting some numbers to the above example, I deployed a research version of a HTTPd (a HTTP along with a TCP/IP stack) on a 28-pin PIC microprocessor in under 3.5 kilobytes of compiled code with a utilization of less than 100 bytes of RAM and a current draw of less than 0.01 amps. It would be impossible to compile the networking code base of Linux, for example, and install it on a microcontroller. Additionally, that layered stack needs a processor that pulls over an amp of current and has megabytes free in memory. Removing abstraction layers is just an optimization that recognizes which portion of the stack can be removed.

3.2.1 Framework for analyzing delayering

The one example above shows that if there is cross-talk between layers of a protocol, it may be possible to compress down the complexity, and therefore the size, of the stack. Future work involves attempting to design a "compiler" for layered protocols in order to create smaller ones as inspired by the work of Mung Chiang, Assistant Professor of Electrical Engineering at Princeton University [Chi04]. It is possible to create a jointly optimal design of congestion control and power control for wireless networks, and questions are whether the same analysis can be performed for other combinations of layers in the ISO/OSI stack.

Chapter 4

Peers do not need servers

In the Internet 0 context peer-to-peer (P2P) means that any two nodes have the direct ability to talk to each other without having to go through an intermediary. This idea manifested itself in chapter 2 in the desire to allow all devices to talk the same protocol so that they can communicate without a "protocol converter" doing all the translation – this allows for a more open network. As future work, P2P in the I0 context also means distributed controllers and algorithms.

As a paradigm, P2P refers to two equal nodes exchanging information directly rather than going through a central broker to get the same information [MKL+]. These systems are naturally distributed as every node has access to information, state, and operations which are not available for every node to have local access to, so therefore its the job of the node to disseminate access to the masses; each and every node has to take on the role of a client when it wants information that it does not have, but it can equally as often have to serve up information to nodes that are requesting it. The main draw to this type of systems is that they address scalability problems for data storage [SMK+01] and computation [WHH+92] in ways that client-server systems cannot because they introduce notions of redundancy and locality into the network[1].

4.1 Client-Server

Client-server systems can be thought of as the relationship between a library and a library patron: the library contains all the information and the library patron goes to request and retrieve the information he or she wants. The library has the centralized information that all the patrons want to get access to, and the patrons all know that the library houses that information centrally.

Another more relevant example is the relationship between a web server and a web browser. The web browser is a "dumb" piece of software that, on its own, cannot perform much. The web server, on the other hand, has the information and the web client has to access and retrieve the information. For example, when

[1]There is also an argument to made about the lawsuit circumvention in a P2P network as opposed to a client-server system, however that is out of the scope for this thesis.

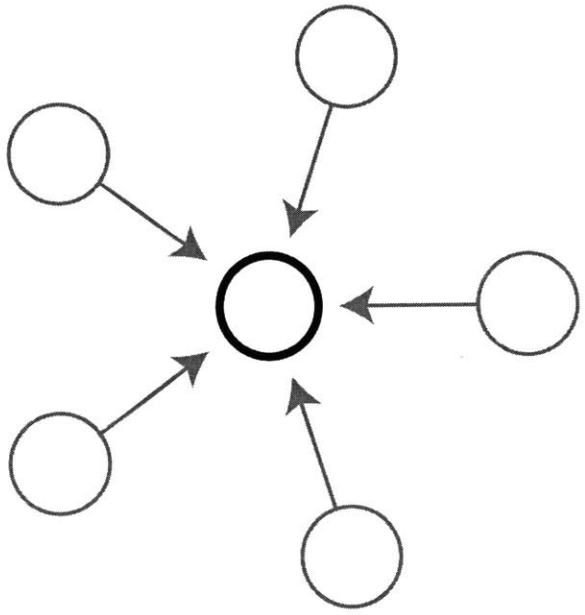


Figure 4-1: An example topological representation of a client-server model. The darkened central node is the server that all the grayed out clients are attempting to access. At no point do the clients contact each other as they all rely on the server.

you wish to access `http://www.cnn.com`, your web browser contacts the server at `www.cnn.com` and asks it for information which it then renders to the screen[2]. Your web browser knows nothing about the information that it is going to be presented, and its only job is to present it.

Lastly, the terminal-mainframe relationship is yet another example of the client-server paradigm. The mainframe houses all the information and all the software, whereas the terminal simply renders that information out to the user. When a user is accessing a spreadsheet or database, the terminal simply sends the user's key strokes to the mainframe which then sends back a series of characters to be displayed on the terminal. No state is saved on the terminal's end of the connection.

It is quite simple to fail these centralized systems. Take September 11th, 2001 – shortly after the attacks on the World Trade Center, people all across the world flooded the Internet with requests for information from all the major news network web sites. Even with their slight attempts (at the time) to distribute their web serving capabilities, every one of the news sites eventually shut down that day under the load. A model similar to the Akamai one of data distribution may have been the perfect solution for that day. Through tricks with the DNS system, every client may have been redirected to a server that is either geographically closer to them (to control the flow of Internet traffic) or to one that is less loaded than the others (to alleviate stress on the servers). A distributed system such as that one may have fared better on that day.

[2]Ignoring any distributed caching system that `cnn.com` may be using. However, even if you do take that into account, it does follow the same model of a client-server relationship, the network is just changing which server you are accessing.

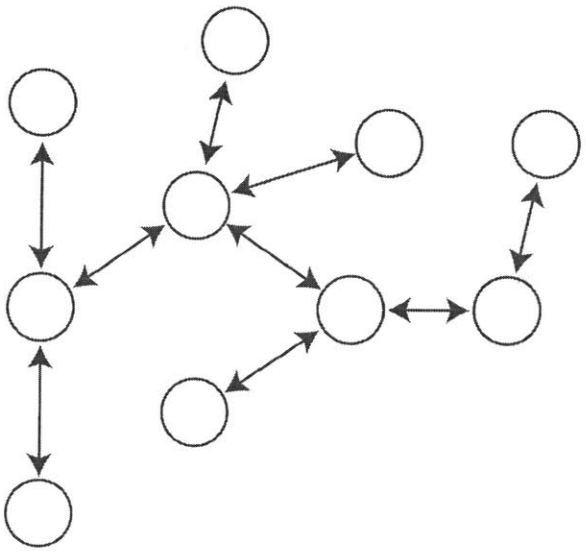


Figure 4-2: An example topological representation of a peer-to-peer model. No node in this graph can be considered as "central", as all nodes talk to each other. It is possible to remove a node that will separate the graph, however that gap can then be bridged by any two nodes.

4.2 Peer-to-Peer

P2P systems, can be thought to be a little more like social systems. Take "tape trading" groups that record live concerts and then make copies to disseminate to other people around the world. No one person has all the live recordings of all the musical shows around the world, and if a person does want to obtain a particular recording, he or she needs to search amongst all his peers in this community to find out who has a copy of that content for duplication. However, the real benefit to designing a system in this manner is that no one person has to keep all the content in a single place – a person only needs to keep content that he or she is interested in, or if he or she is feeling benevolent then he or she may keep some extra content around for the good of the network. Each node is of a common status within the network, and each has to interact with each other.

Gnutella is an example of a software P2P system that is deployed and running on the Internet[3]. In the Gnutella network, each node creates a bi-directional connection to a number of other Gnutella nodes. When a node initiates a search of the network, that search message is propagated to all nodes that it is connected to, and so-on (with some consideration taken into account for the time-to-live on the message, and some memory to prevent a search message from looping on the network). If a node has a piece of content that matches the search, then that node informs the originating node of the "hit". The originating node may then choose to contact that node to ask for a download of that file.

[3]Most people also associate the original Napster as a P2P system, however they are partially incorrect. It is true that the peers directly contact each other to deliver files to each other, however the search on the Napster network was centralized: all nodes would publish a list of the files they are sharing to a central Napster node to which any node could search to get a pointer to the node that is hosting the file. From that pointer, the node could initiate the direct file transfer.

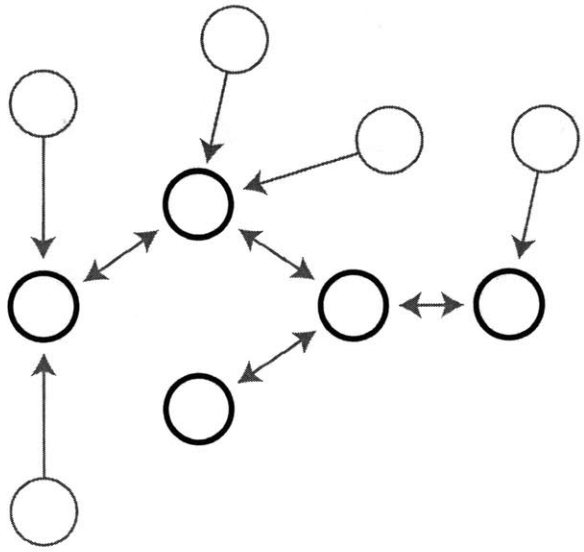


Figure 4-3: An example topological representation of a hybrid client-server and peer-to-peer model. As in figure 4-1, the darkened circles are the servers, however now they are acting as peers amongst each other. The grayed out circles are still clients, however, who are communicating directly with the servers in the network.

It is worth also noting that P2P systems also come in a variety of hybridizations (see figure 4-3) [YGM01]. For example, consider the electronic mail systems on the Internet: an e-mail client is usually setup to send its mail via a single simple mail transmission (SMTP) server. That server, however, is rarely the destination of that e-mail, and it is that server's responsibility to parse the destination of the e-mail, contact the appropriate SMTP server, and deliver the message. Those two SMTP servers are "peers" in this network, but the originating e-mail client is a client to those servers.

4.2.1 Client-server vs. P2P

Both client-server and P2P systems have their benefits. In the client-server camp, the most notable one is that client-server systems are quite simple to engineer. All the intelligence and data are in one place, and it makes it simple to make inferences and decisions. The clients only need to see is a rendering of what is going on in the system and have no authority over any of the data. However, this system is also quite vulnerable to attack and failure: by simply failing the central server, the entire system becomes unusable; there is no redundancy in this system. Also, a client-server system may be inelegant for particular problems. For example, take a situation where a client wants to affect a change of state onto a different client in the system – in a client-server system, the change needs to be "routed" through the central server first. If these changes are happening fairly rapidly with may clients attempting to affect many other clients, the central server becomes the bottleneck in this system. If that central server becomes overloaded, or fails, then no client can affect another.

While this may all seem to be a resounding endorsement for P2P systems, keep in mind that P2P systems can be slightly more difficult to engineer properly. There are distributed consistency problems, the data is

not located in one place so therefore an efficient distributed search mechanism is required [SW02], there are partial failures that can happen in the network, and the list goes on. However, the major benefit is that there is no *single* point of failure in the entire system: if a single node goes down, it will take down any state and threads of execution that are associated with it. The rest of the network will remain up and running.

Going back to the ever present light switch and light example: consider a "smart home" that brokers all the messages in the house. If that server ever goes down, then the entire house stops functioning. This is very far from the status quo of how light switches and bulbs operate in a house today as there is no single point of failure in a home today (save for the electrical system). If the smart home has the switches and bulbs functioning as peers on the home network, then there is no single point of failure that can disrupt their communication. If, by chance, the light switch or bulb breaks, then only those two nodes are affected and the rest of the network can proceed as though nothing has happened.

Simply stated, if constructing a network of Internet Protocol enabled devices, the devices can and should be designed to intercommunicate directly with each other to reduce the number of external dependencies in the network.

4.2.2 Message Passing Algorithms

Future work in Internet 0 encompasses a study of graphical message passing, and the applicability of these mathematical models to networking. The basis of this work are factor graphs and the sum-product algorithm [KFL01] where the probabilistic relationships between dependent variables are indicated, and that is used to pass probabilities through the network until a solution is relaxed out. This algorithm has the natural ability to decompose a problem into its hierarchical parts that exhibits decentralization and locality – exactly the same two properties of a P2P system. It is the hope that these types of algorithms will allow a network to make decisions and compute in ways that are currently only done in centralized systems.

4.3 Hierarchy

Although the infrastructure of Internet 0 is designed in a peer-to-peer fashion, this does not mean that there is not a place for a centralized server, or supernode, somewhere in the network. For example, Google provides an invaluable resource to the WWW, without which, the WWW would continue to function, however would not be as useful as it would be nearly impossible to locate information; the centralized indexing that Google provides adds value to the network, but without it the network would continue to operate. More generally, these supernodes can provide high level functionality that cannot be obtained from a single device in the network. As said many times before, the network should not require centralized nodes, however it should be recognized that centralized nodes have a place as controllers and data aggregators. These supernodes fill a niche that a single node in a distributed network cannot.

4.3.1 Data aggregation

Each individual node in the network may know about its particular state and history, however no node should be expected to know the detailed state and history of any other node, let alone the entire network. For example, each node in a distributed array of sensors may be producing a continuous stream of data that does not make much sense without the "context" of the other nodes in the network. While each node could listen to all the data in the network and individually infer the larger view of the data set, it seems redundant and perhaps computationally impractical to do so. This applies to sensors that may be on a battlefield collecting information about the terrain and the enemy, to amperage meters in different locations on a campus: individual pieces of data may be interesting, but no decision can be made without seeing all the data simultaneously.

4.3.2 Centralized control

Centralized control follows directly from data aggregation (section 4.3.1). There are cases when individual nodes talking to other individual nodes makes sense, such as the example of the light switch talking to the bulb. This example can even be expanded to a daisy chain of events such as having the talking to the bulb who then relays a message to a thermostat which then determines whether the room is too hot or too cold and activates the HVAC system for the room. Of course, this daisy chain of events does not necessarily scale.

Additionally, consider synchronization issues. For example, any life threatening piece of machinery that may be controlled with an Internet 0 data-bus. Each individual portion of the machine could be made to operate and make decisions on its own, or the portions of the machine could be designed to accept instructions from a centralized node on-board that is issuing commands. The latter is more preferable in the case that the machine needs to be shut down quickly in an emergency situation. A single command to the central controller could cause it to disengage the entire machine whereas if the machine is built with distributed controls then a command would need to be issued to all controllers on board for the machine to shut down. Centralized synchronization is easy – distributed is not [GC93].

4.3.3 Hierarchical organization

Lastly, hierarchical control follows from the hybridizations of peer-to-peer systems (shown in figure 4-3). Attempting synchronization of clocks around a network [Mil94] requires a tree of computing nodes simply as an optimization – overwhelming a single node with time requests would constitute a distributed denial of service attack. Therefore to mitigate this issue, the Network Time Protocol (NTP) constructs strata of servers. Stratum 0 servers are those machines who actually have physical access to the time, whether they be attached to an atomic clock, or synchronized via a GPS connection. While the majority of the network would like to connect to these servers, they simply are not allowed to do so to protect those nodes from the vagaries of the network connection. Instead, the stratum 0 nodes are connected to a stratum 1 node who in turn connects to many other stratum 1 nodes (for synchronization) and a myriad of stratum 2 nodes (for distribution). From

there on, its open season.

The two items of note from these types of systems is that while a hierarchical organization alleviates load on the "supernode", it does introduce a variance that goes up as you go further and further away from the root [Min99]. An entirely P2P system (section 4.2) may experience an overload of certain peers, however it will not have a problem with false information[4]. Issues of trust in the network become very apparent, and perhaps the only way to deal with it is to create a trust structure in the system [Zis97] as this type of system will become very useful in architectures where individual nodes provide valuable information, but not enough processing or memory to provide network connectivity to the entire world.

[4]Unless the node with the information is deliberately lieing.

Chapter 5

Physical Identity

One fairly unique and high-level concept for an Internet 0 network is the notion of physical identification and access to the devices. One of the original requirements for the creation of a unified global network of computers was to enable access to resources no matter where the user is physically located – unfortunately that exact feature may doom Internet 0 type networks. As Internet 0 devices may be used to control critical and non-critical members of an environment, a different paradigm for identification and access is necessary: physical in cooperation with the online.

There are certain types of interactions that one would like to make with physical access to the device or the network. Again, with the light bulb and the switch, one simply wants to be able to turn on the light bulb via the switch. You should not need to manifest a computer to make that happen – the light switch should be the conduit to the network. But also, there are security concerns that can be addressed with physical access. One can lock out access to a network node, simply because he or she does not have a means to physically get to it.

Before interacting with a device, however, one must first identify it. Identity is a subject that grows without bounds when it comes to computers. For people, identity is relatively easy (but not entirely) – for computers, this can be a very daunting task. How does a computer know which other computer it is talking to? How does a computer which person it is talking to? How does a person know which computer he or she is talking to? And, how does a person, via a computer, know whom he or she is talking to? Computers need to deal with identity on many different levels: the network level, the physical level, and at a logical level. Each of these types are distinct, however, when used in different combinations, the identity becomes more than a sum of the individual parts.

Once two computers can identify, then we can add a physical aspect of proximity to the equation.

5.1 Identity

As mentioned above, there are three types of identities to assign to an Internet 0 device. Network identity involves the IP address of a node, physical identity is the (mostly) unique identifier for a device, and finally there is the logical identity which is the relationship between an object and the rest of the world.

5.1.1 Network

Network identity is type of computer identity that most people are familiar with. Computers are assigned an Internet Protocol address either directly by a network administrator, or indirectly through a DHCP server – and this identity is how other computers from around a network or the Internet can use to address messages to this computer. This IP address on its own cannot be used as an identity because that address is assigned to that computer solely based on its location within the network.

IP addresses are not assigned permanently to computers, and when that computer moves, it may get assigned a new address. If a laptop is taken from the owner's office network and connected to the owner's home network, for example, it will inevitably be assigned a different IP address[1]. This occurs because the computer has changed its location on the Internet and addresses are assigned to computers to make global routing simpler – classless inter-domain routing (CIDR) [FLYV93] relies on IP addresses to be handed out in a hierarchical fashion. Before CIDR was introduced, IP addresses were handed out in a static manner, meaning that when an organization did physically move or change Internet Service Providers (ISPs) they could take their IP address block with them. This, unfortunately, has the side effect of forcing routers to keep information on every single IP block, and possibly every single IP address for routing purposes. CIDR means that routers can instead keep information about that hierarchy instead; pre-CIDR each router on the planet would need to keep information about where a given computer is, but post-CIDR, a far away router only needs to know a "broad" idea of where a computer is globally, and the closer routers will have more details about the location. The far away routers are relying on the closer routers to know more.

Another reason that the IP address may not be the best identifier for a computer is partly because there is a shortage of IPv4 addresses on the planet. This problem manifests itself for a couple of reasons: the first being that there is only enough addressing space for 4,294,967,296 computers (2^{32}). While that may seem like a large number, the real underlying problem is how those addresses are assigned. MIT, for example, has a class A block – the entire 18.0.0.0/255.0.0.0 address space – giving it enough addressing for 16,777,216 computers (2^{24}). An organization like MIT does not have over 16 million active devices, so the rest of the world is at a loss.

Another reason is due to the pricing structure for access to the Internet. ISPs traditionally charge per IP address or per IP address block instead of paying for access to the Internet. While this benefits some (those

[1]There are, of course, work-arounds to this. There are a few mobile host protocols [MS93] and also virtual private network (VPN) solutions that proxy IP at different levels to allow a computer to have a "static" IP address. They do use packets to go bouncing all through the Internet, however.

mainly who transfer a large amount of data over their network connection), it penalizes the average user who may have more than one network device in their possession, but only use ones at a time. The common solution for a home that has broadband access is to buy a "home router": a combination firewall, DHCP server, and NAT [EF94] box. The combination of the DHCP server and the NAT box allows the router to obtain the one IP address assigned to that broadband connection, and then assign globally unroutable addresses to all the machines on the inside of the network. Whenever a packet destined for the outside world is transmitted from the inside network, the gateway rewrites the packet to come from the gateway's IP address. When a response comes back, the gateway can again translate that packet for the internal network. This ISP pricing structure has the side effect of causing all the machines on the inside of the network to be directly unroutable to the outside world (as they are all identified by the same IP address to the outside world) and also use IP addresses inside the NAT that are used in other NATs around the world.

These two reasons make IP addresses, on their own, be an unsatisfactory identifier.

5.1.2 Hardware

In computer networking, a media access control (MAC) address is the physical identifier of the network interface. The MAC address is designed to operate at level 2, and provides a guaranteed unique address that is used in Ethernet [RP94], 802.11, ATM, and a other networking technologies. This guaranteed uniqueness provides the ability for two network cards to communicate with each other without any issues from layer 3, where addresses (IP) are not guaranteed to be unique. However, maintaining this guaranteed uniqueness is an administrative nightmare.

In order to obtain a block of addresses to utilize, one has to contact the IEEE registration authority and request a 48-bit block. The IEEE makes sure never to assign the same block of addresses to two different parties, at the cost of having to purchase either an "organizationally unique identifier" or an "individual address block" at the cost of $1650 and $550 respectively. The cost associated is with the maintenance of the IEEE database and to limit the number of people making frivolous requests for MAC address blocks. The side effect is that this does lock out smaller developers and experimenters from creating network interfaces to guarantee a protection which probabilistically does not need to be enforced[2].

This makes the MAC address methodology might make an appropriate identifier, save for the prohibitive cost and administrative overhead to do so.

5.1.3 Logical

Logical identity solidifies notions of "my computer", or "the light switch by the door". Issues of trust and relationship are wrapped up in those two statements. A user trusts and expresses a relationship with the

[2]In 1998 the Certified Software Corporation (CSC) attempted to purchase a block of MAC addresses with the purpose of reselling smaller hierarchical chunks to prototypers and low quantity developers. Their attempt was stopped by the IEEE under the premise that the organizationally unique identifier that was sold to the CSC was to be only used in products manufactured by the CSC.

data and threads of execution on his or her computer, and the logical identity of the light switch states its relationship with any other appliance in the room or in the building. Unfortunately, due to the nebulousness of this type of identity, there is no computationally algorithmic way to assign and maintain them. There is a real notion that is being expressed in this type of identity that is missing in many computer systems – relationships.

Currently, computers have no notions of long term relationships. The best attempt at one is the recording of the public keys of SSH servers that a computer connects to. Recording these keys helps prevent later man-in-the-middle attacks. However, this relationship does not reflect any changes in the system that may occur. When one first connects to another machine, the SSH public key is recorded, and that is held for granted every time that machine is connected to again.

Also, computer networks have no notions of physical proximity. There is nothing in the IP layer or in the hardware layer that specifies spatial positioning. Given that a subnet may be as small as room, or as large as a continent means that nothing can be really inferred from the network address. Hardware addresses are also not useful as they are effectively physically spread randomly around the world. A logical identity of "the light switch in the room with that light" does say something, however.

5.2 Generating Identities

Because all Internet 0 devices do need an identity of sorts, there needs to be an algorithmic way to generate them. One hardware address is needed for each device to as provide a nearly-unique identifier for this device, and then at least one IP address is necessary so that the device can get connectivity.

5.2.1 IP

IP identities cannot be given to devices at manufacturing time because, as mentioned above, CIDR requires that devices be assigned IP addresses according to where in the hierarchy they are located. However, because as stated in chapter 4, an Internet 0 network cannot assume the existence of a DHCP on the network, so therefore an IP address needs to be generated at runtime.

Internet 0 borrows IP address allocation from the zero configuration specification [SAA03]. When a device first comes online, it issues a DHCP request to the network. If the server is heard from, then the device can assume all the information given to it. However, if no server is heard from (after a series of requests), then the following needs to occur so that the device can assign itself a link-local address:

1. Choose a random address from the 169.254/16 subnet (except any address in 169.254.0/24 and not in 169.254.254/24),
2. send an ARP "who-has" request for the chosen IP address, and
3. if an ARP response is heard from, repeat from step 1.

4. Otherwise, send an ARP response claiming the address, then

5. defend the IP address by responding to ARP requests for this address.

Whenever the network interface of the device is reset (rebooting, power cycling, awakening from sleep), then the above procedure must be repeated. As an optimization, the address could be written to non-volatile memory, and upon reset skip step 1 and start immediately at item 2.

5.2.2 Hardware

Again, hardware addresses cannot be assigned at manufacturing time as it becomes infeasible to maintain a centralized registry of any considerable size, and still allow experimenters to participate in the network. For that reason alone, random hardware address generation is needed.

Fortunately, generating these hardware addresses are simpler than generating IP addresses (section 5.2.1) – simply choose a run of 64 iid bits. There may not even need to be a necessity to defend this chosen hardware address as the probability of two 64 iid bit sequences colliding is 1 in 1.84×10^{19} (where as, the probability of two IP addresses in the zero configuration subnet colliding is about 1 in 6.50×10^4).

5.2.3 Random number generation

The most difficult portion of both address generators mentioned is the creation of the random number sequence. A linear feedback shift register is the first type of generator that comes to mind – given m bits of state, the LFSR can output a repeating sequence of 2^m pseudo-random bits (after those 2^m bits are outputted, the exact same sequence will be outputted again). The LFSR needs to be initialized first, however, with a seeded value for the first tap sequence. On a personal computer, it may be ideal to stuff the tap sequence with the current time, however a microprocessor does not have that luxury. It may be possible to, instead, leave those bits of state in an uninitialized state on power-up; whatever values are electrically present in the registers can be taken to be the random start up state. Another possibility may also be possible to sample bits over the network interface (even if it is not initialized) and feeding that into the taps. If the taps are not initialized into their ideal start state, then the length of the sequence generated by the LFSR will be shorter than 2^m.

Another, more tempting, proposition is to use a physical process, such as a physical one-way function [Pap03] for the random number generation – either as the hardware address or as a seed for a LFSR.

5.3 Access

Physical access, along with identity, changes aspects of the scenario. While it may be possible to properly identify a device over the network, it may not always be ideal to allow programming over a network. It has already been stated that servers should not be necessary to interact with the network (chapter 4) – physical

access to the device follows that requirement by allowing people to program the network without a server. Physical access also changes aspects of security with respect to the network.

5.3.1 Programming paradigm

It may be helpful to work through a scenario to demonstrate a physical programming paradigm. One could imagine a few different network topologies, the first having a centralized broker with whom all the light bulbs and switches interact with. From there it is easy to change any associations, as one just has to modify the database on that one broker. This scenario is obviously rejected as it does not follow any of the framework that has already been laid out in chapter 4.

The second scenario has all the light switches and bulbs interacting with each other directly. There does not exist one place in the entire network that has access to all the associations in the network. Because of this, one has to deal with the light bulbs and switches directly. Each bulb and switch could export a network API that allows you to set the association over the network. Unfortunately, unless some type of authentication scheme is put into place, this means that anybody who has access to that network (and therefore, possibly anybody on the Internet) can control which switches are talking to which bulbs. It also has the added inconvenience of requiring that a third network appliance be used to modify the network. This again, is going to have to be rejected.

For a third situation, consider a network as described in scenario two, except there is no network API for establishing device associations. Instead, to cause an association one needs to physically access the switch and the light bulb and possibly press a toggle switch, or perform some type of interaction with both that coerces them to interact with each other over the network; each one requests the other for an association. This type of physical programming does away with the unfortunate side effect of allowing associations to be created over the network and also does not require another device to make the connection.

5.3.2 Consequences for security

The third situation above (section 5.3.1) also raises interesting points about securing these networks. While future work does include the development of a secure way to interact with these small and embedded network devices, physical access to the device does and will continue to play a very important role in the network.

Take an elevator that is controlled via an Internet 0 bus: while it may be convenient to display information about the elevator over the network, at no point do you want random people to be able to modify the state of the elevator from any point in the network. The simple solution is to hide the access point behind a lock and key and if third party does not have possession of that key, then nothing can be done to affect the state of the device, even with network access to it.

Another interesting aspect are future cryptographic protocols which involve physical identity. The current proposal for security involves symmetric encryption so that two devices may prove to each other that they

know the same shared secret. However, most encryption mechanisms (whether they be public/private or symmetric key cryptography) are agnostic towards the "distance" between the two parties. Each party could be attempting to prove themselves to the other from across the room, or halfway across the planet. A possibility is to include a short-range I0 transceiver on a device for identification purposes, that requires the two devices that are interacting with each other to be within close proximity. This changes some of the assumptions that can be made in the design of the cryptographic protocol, because any attempts at identifying a third party when they are not physically close by can be simply ignored.

Chapter 6

Big Bits

Most development in networking technology is dedicated to going as fast as possible because it means saturating all available bandwidth on the channel. For the Internet, hardware research is devoted to faster-than-terabit Internet 2 links while software research delves into saturating those links [JWL04]. What is being forgotten, however, are two crucial points

1. A light bulb does not need to watch video on demand, and

2. there are many hidden costs of pushing bits quickly.

These two points are the motivation to slowing down the speed of the Internet 0 network to be "fast enough" and causing the bits in the media to grow large. Going slower means less cost and complexity in the network – in the end it means more functionality for Internet 0 devices.

6.1 Fast-enough networks

Pushing a bit very quickly through a network means cost of hardware, costs of installation, and costs of maintenance (see section 6.3). A slow bit, however, is very cheap. Knowing this, we can create a framework that will help us determine a reasonable upper bound on the bandwidth necessary for a home or building. First consider the traditional light bulb and switch – the only intercommunication needed for the two involves the transmission of a message from the light switch asking the light bulb to turn on. As shown in chapter 2, a TCP light switch and light bulb may require as many as five packets to be interchanged to confirm that the light has indeed changed state (as this TCP light switch and bulb pair will require more bytes to be transmitted than its UDP counterpart, it is reasonable to use it as the upper bound). According to table 6.1, it takes approximately 240 bytes to change the state in a light bulb[1].

[1] Of course, this data is only necessary to *change* the state in the building. In order to maintain the status quo, no packets need to be shuffled around.

Table 6.1: The packets, along with their associate sizes, that need to be transmitted between a light switch and a light bulb to request a change of state.

Packet contents	**Packet size**
The initial SYN packet from the light switch to the light bulb. This requires 20 bytes for the IP header, and then another 20 bytes for the TCP header).	40 bytes
The light bulb needs to respond with a SYN/ACK TCP/IP packet.	40 bytes
The light switch can then ACK the SYN/ACK along with the request for the light to turn on. Again, the TCP/IP header is 40 bytes, but then there is the data. Assuming a HTTP GET is used, that will take, conservatively, 20 bytes.	~60 bytes
Now the light bulb needs to acknowledge the reception of this packet, send data that confirms that the light bulb has been turned on or off, and also set the FIN flag to terminate the connection. 40 bytes for the TCP/IP header, and then another 20 bytes for the HTTP response.	~60 bytes
Lastly, the light switch needs to respond with the FIN bit set in the TCP header to close its end of the connection.	40 bytes

As a desired goal would be to keep within the expected requirements of a light bulb and a light switch, it would be ideal that when the switch is pressed that the light turn on "immediately". Given that the human reaction time can be measured in tenths of a second, an appropriate time requirement can be stated as "within a tenth of a second of a light switch being pressed, the light bulb should begin its power-on sequence". 240 bytes need to be transmitted through the network within a 0.1 seconds, therefore that sets the network speed to be 18.75 kbps[2].

The probability of two people switching the state of two independent light switch and bulb pairs on the same Internet 0 network is quite small, however if that did occur, there are number of different packet interweaving that could occur (approximately 1000 of them) – however, we need only consider the worst case: one light pair's entire set of packets is processed on the network before another pair's. Meaning that one switch will turn on within a tenth of a second, while the other will turn on within two tenths of a second. If that is unacceptable, then simply doubling the speed of the network will allow two switches to simultaneously turn on within one tenth of a second.

This framework is the appropriate way to determine the speed necessary on the network. First, the bandwidth needed for a single set of interactions to occur must be accounted for. Then the probability of a collision of two of those sets of events must be considered and the speed of the network scaled up accordingly.

[2]A slightly different argument states that only 140 bytes need to be transmitted for the light to turn on, as the 60 bytes from the light bulb acknowledging the change state request, along with the 40 bytes from the switch closing its end of the connection happen after the light turns on. Following that computation through means that a network speed of only 10.94 kbps is needed to turn on a the light bulb within an average person's reaction time.

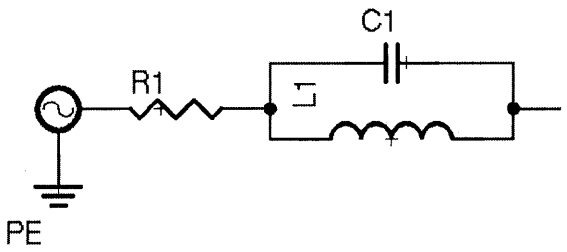


Figure 6-1: A wire, modeled as a resistive element that also has a parallel capacitance and inductance, with a signal being transmitted over it.

6.2 Bits have size and shape

If you consider that bits travel at approximately $\frac{2}{3}c$ in copper and in fiber optic cable, a given bit rate translates to a bit having a particular size on a network as dictated by

$$\frac{\frac{2}{3}c}{r} = \text{length of a bit} \tag{6.1}$$

where c is in meters per second and r is the rate that the information is being transmitted in bits per second. This means that on a one gigabit network each bit is just under 20 centimeters long and on a 18.75 kbps link as described above (section 6.1), each bit has a size of just over 10 kilometers. A building who may have its longest link at 100 meters can afford a data rate of about 2 megabits per second.

When transmitting at a bit rate that puts the size of the bit significantly smaller than the size of the network, then one is operating as if one were in the "near field"[3] – for the above example, that means operating at a megabit a second over distances of less than 200 meters. Intuitively, this means that (chances are) at any given time the receiver is seeing exactly what the sender is transmitting. All multi-path reflections can be ignored because during that time interval the entire system has equilibrated out. To be a little more concrete, take for example a wire that has a step function being transmitted across it – approximately during the time the signal stays constant (at either high or low) the receiver sees the same constant signal.

If we update the model of the wire to have a given resistance, capacitance, and inductance (see figure 6-1), then a few complications arise. Figure 6-1 has a transfer function of $H(s) = \frac{sL}{s^2CLR+sL+R}$ across the parallel capacitor and inductor. Given a step function as an input, the response over the capacitor and inductor ends up as

$$V_0(t) = 1 - (\cos\frac{\sqrt{4CLR^2 - L^2}}{2}t + \frac{L}{\sqrt{4CLR^2 - L^2}}\sin\frac{\sqrt{4CLR^2 - L^2}}{2}t)e^{\frac{-Lt}{2}} \tag{6.2}$$

This system settles down rather quickly after the step response; the influence of the exponential decreases by a factor of a hundred after $t > \frac{9.2}{L}$. If the input signal is stepping between 0 and 1 rather slowly, then

[3]The Near-Field Forum (NFC) recently founded by the Nokia Corporation, Royal Philips Electronics, and Sony Electronics use this term in a slightly different way to mean communication via electromagnetic coupling. For this thesis, it is being used to mean simplified communication.

for the majority of the time (after the transient portion of the equation has died out), the system equilibrates out to what the transmitter is transmitting. However, if the sender is transmitting very quickly (and therefore, switching between the 0 and 1 input fairly rapidly), then a qualitative mess is occurring on the wire.

To further complicate the situation, if the wire is not impedance matched on both sides then a mismatch occurs on the amount of energy being sent down the wire. If the source impedance does not equal the complex conjugate of the load impedance, there will be an inefficient power transfer between the two – meaning a reflection of a portion of the transmitted energy back to the source. These spurious "echos" inside the transmission will possibly confuse the receiving node.

6.3 The cost of high speed bits

The theory laid out in section 6.2 all translates to physical implementation issues and costs. Having "small" bits on the wire requires agile radios that can cope with high speed transmissions and the transients that are caused by the impulse response of the wire. Ethernet transceivers today cost $10 at unit cost, which can be more than the cost of the device that we are attempting to network.

Along with the proper radios, the proper cabling is required for transmission. Transmissions are usually sent differentially over twisted-pair cables to reduce coupling due to mutual inductance – hence Ethernet's use of category 5 cable, which consists of four twisted pairs of copper wire supporting up to 100 MHz frequencies and speeds up to a gigabit per second. The complexity of the wire, and the tolerances to which it needs to be manufactured are reflected in its monetary cost. Additionally, these wires have to be very carefully impedance matched on both ends, otherwise reflections and energy loss are bound to over power the system.

The cost of these high speed bits unfortunately do not stop at the price to pay for the radios and the cabling. Because of the strict requirements on the transmission medium, these cables cannot be arbitrarily split to create random topologies; the cables are meant to be used only for point-to-point links and cannot be spliced into two wires in the hopes that it will work. In order to split a wire, an impedance matched radio is necessary at every termination, and hence the need for hubs and switches at every desired split. Not only does choosing the right hub or switch require experience and knowledge, but laying out the wire into the right topology for a building does too. Equipping a building for a high speed wired network is no longer a blue collar job, however many buildings are having the network integrated into their building design meaning that a series of networking engineers are needed during the architectural stage, and also at the building stage to install the all the proper gear.

All of the above changes when operating a lower speed network. Radios become simpler because they do not have to operate in a domain where transients and echos are occurring, as they can just wait until the network settles down; hardware costs go down as radios and cables become cheaper. Also, because we are not as worried about impedance matched wires, it is possible to simply split a wire wherever necessary. An entire building can be wired for Internet 0, low speed networking, simply by running a series of wires, and

splicing them every time a split is necessary. No hubs or switches necessary.

Chapter 7

End-to-end modulation

Modulation schemes attempt to represent information for transmission through a given medium. As Internet 0 devices can be connected to any given number of media, a more traditional train of thought would require a medium dependent modulation scheme so its specific properties, such as the pass-bands and time delays, can be properly and rigorously accounted for. In the spirit of IP to the leaf nodes (chapter 2), however, another possibility emerges: use one modulation scheme across all possible media.

The same arguments that are applied to unifying different network protocols into IP can be considered for using a single modulation scheme in an "end-to-end" fashion. Notably, it means that the representation of information is independent of the carrier that is transmitting it. If information needs to change what medium it is traveling over, there needs to be no complicated intermediary receiving and demodulating the data on one side to then only remodulate and transmit it again on the other side – instead, the data can be simply passed along; all that is required in the middle of the network are "dumb" nodes that can amplify a signal. Similar to how no intermediary hosts on the Internet need to process the data in the packet itself, they simply need to forward a packet along, the intermediate hosts in an end-to-end modulation scheme need only pass along the modulated data without ever actually processing it.

A comparison can be made to Morse code. This modulation/protocol can be used across any number of media: electrical charges over long wires strung across the Midwest, smoke signals, banging on the hull of boat, to flashing a flashlight. In fact, a repeater is possible just by mimicking the dots and dashes that are observed by the intermediary; at no point does a repeater have to demodulate the information and then remodulate it. In both Morse code and in the Internet 0 end-to-end modulation scheme, the information is represented the same way across a variety of media.

7.1 Modulation

Modulation is defined as the process of representing information for transmission. For example, amplitude modulation (AM) takes the information to be transfered, and encodes it into the amplitude of a carrier frequency. AM is a very simple modulation scheme that functions very well over media that have an easy ability to transmit variances in the amplitude of a signal, unfortunately most naturally occurring noise is white (an additive Gaussian white noise signal), and AM modulation is going to be very susceptible to interference from it. On the positive side, it is very simple to implement both the transmitter and the receiver of the signal as an oscillator will allow you to modulate the signal, and a phased locked loop will allow for demodulation. This simple example illustrates a few key points of what needs to be taken into account when selecting a modulation scheme: the capabilities of the medium being transmitted over, the type of noise that the signal is bound to encounter over the channel, and the complexity of the transmitter and receiver pair needed to establish communication.

Unfortunately optimizing those usually means that different modulations are chosen for different media: IrDA can be used for IR transmissions, FM for radio links or even ultrasonic links, differential serial for wired links, and the list goes on. As mentioned above, traditional thought leads to very different modulations that is dependent on the medium.

7.2 Impulse radio

The promising exception to the "different modulations for different media" rule is impulse radio [WS98]. Commonly known as "ultra-wide band", this scheme was originally designed for radio links, but it is tempting to imagine an extension of this scheme that functions across any media. Impulse radio uses pulses of very short duration whose energy is spread very thinly from DC to the speed of the click itself (a nanosecond click spreads from DC to 1 GHz). Assuming that the onset of the click can be very accurately measured, careful positioning of the click can allow for many simultaneous users of a channel while being impervious to multi-path reflections [WS00].

A click is effectively measuring the impulse response of the medium – the transmitter is "hitting" the medium with a very short, but very large, amount of energy that the medium rings with. Transmitting a click over a medium can be done very trivially – for an RF link it simply means emitting a short burst of energy through an antenna, for an IR link it means blinking an LED, and for a water link it means inducing a tidal wave. Conceptually, reception is rather straight forward and requires a match filter to catch the impulse on the receiving end[1]. Even a microcontroller has excellent time resolution, and therefore the output of that match filter can be fed directly to a microcontroller who can then precisely position the impulse in time.

[1] It may be even simpler. As the pass-bands of the medium, or of any of the media that the impulse may have traveled through, is not known a priori, it may be difficult to construct an exact match filter to catch the input. Assuming an AWGN (additive white Gaussian noise) channel has a given amount of energy, a receiver may simply be able to look for a sharp increase of energy in the channel during a given time interval.

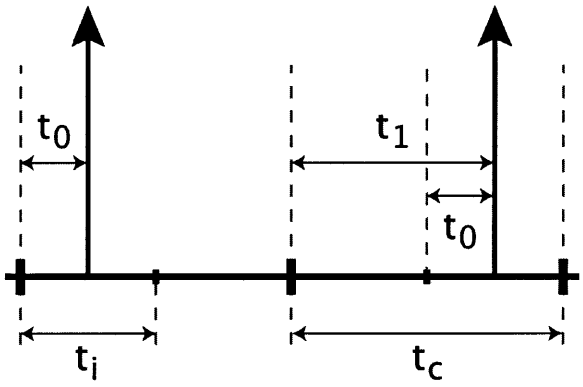


Figure 7-1: The placement of a zero bit and a one bit. $t_i = 2t_c$ where t_c is the length of a click interval and where t_i must be greater than the impulse response of the media transmitting the data. t_0 is the time after the start of t_c and $2t_0 = t_i$ to expect a 0 bit and t_1 is the time to expect a 1 bit.

Noise rejection is performed both by having the transmitters insert enough energy into the medium to get above the noise floor, but mostly by careful positioning of the impulses onto the channel. Traditional impulse radio uses a pseudo-random time hopping sequence [LMLL03] to allow for noise rejection and multi-user access – if the impulses are only allowed to be transmitted during very particular, and pseudo-random, slots, then it is very hard for noise or other transmitters to catastrophically interfere with the transmission.

7.3 Internet 0 Clicks

Internet 0's "clicks" are directly inspired by impulse radio (section 7.2). Instead of utilizing a spreading code, Internet 0 relies on self consistency in the placement of the clicks. Figure 7-1 defines the location of a 0 and 1 bit given a particular interval. Each bit is divided into two slots in a Manchester-like encoding scheme. t_c is the "click interval" and t_i is exactly half that ($2t_i = t_c$). t_i is bounded at the lower end by the length of the impulse response of a medium as that allows for the medium to settle after an impulse has been sent; if t_i is not bounded by the impulse response, then it may be difficult to determine the start of two successive clicks. The 0 bit is to be transmitted precisely at the center of the first t_i in a t_c, and the 1 bit is to be transmitted at the center of the second t_i.

An entire byte is framed by a start and stop bit sequence which is a t_c long and where there are *two* clicks, one occurring at t_0 and the other at t_1 from t_s. This byte framing has the characteristic that it allows for the sequence to be self clocked and therefore operate at any timescale. When receiving the impulses, the distance between the clicks in the start byte defines the length of t_i (and therefore the length of t_0 and t_1), and also the start of the entire byte sequence. Another aid is that the receiver knows that it expects a start sequence, followed by 8 bit sequences ordered with the most significant bit first, followed by a stop sequence. If any of the expected t_is are empty, then an error has occurred and the entire byte can be rejected.

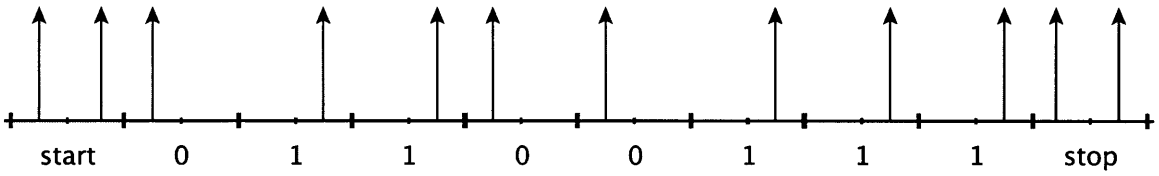


Figure 7-2: An example byte sequence encoding 0b01100111 (103) – pre- and post-pended are a start and stop sequence which involves encoding two clicks within t_c for framing and self clocking.

7.3.1 Decoding click sequences

Given a timeline of click events, each click timestamped, an exhaustive $O(n^2)$ search can reveal all the encoded bytes:

1. Start at a specific click (with a timestamp of n), and
2. pick another following click (with a timestamp of m where $m > n$). Consider n and m to be the start sequence – this means that the $t_1 = m - n$, $t_i = 2t_0$, $t_c = 4t_0$, and the start of the byte is at $n - t_0$. Call that t_s.
3. From there, look for eight data bits (bits 7 through 0 as the bits are ordered MSB) such $b_n = t_s + (8 - n)t_c + t_0$ or $b_n = t_s + (8 - n)t_c + t_1$. All eight must be present.
4. Then look for the two stop bits at $t_s + 9t_c + t_0$ and $t_s + 9t_c + t_1$. Again, both must be present. If they are, then a byte has been located and decoded.
5. Continue for all n.

Assuming the impulses are being received by a device with good time resolution, this specification has the property that any spurious noise (manifested by extraneous clicks) can be trivially rejected because probabilistically they will be inconsistent with the byte sequence (as shown in figure 7-3).

It is possible, however, that given a sequence of bits with a particular t_c created from a sequence of bytes could have larger patterns of bytes (a t_{c_l} where $t_{c_l} > t_c$) which are entirely self consistent – figure 7-4 shows this quite clearly. In simulation, one transmitter encoded a string of 100 bytes at a t_c of 100 (unitless), however when the receiver decoded the bit sequence it not only recovered the entire original sequence, but also (on average) another 900 other bytes all with click lengths ranging from 200 to 3500.

There are a few strategies that can be used to discard those extraneous bit sequences. The first is to use a spreading sequence to carefully place the positioning of the beginning a byte sequence. By positioning the byte sequences in such a manner, it is possible to eliminate the need to search through all possible n as stated in the algorithm above. There is still a possibility for spurious bit sequences, but the probability is greatly diminished. Producing the sequence on the transmitter can be implemented in software and the

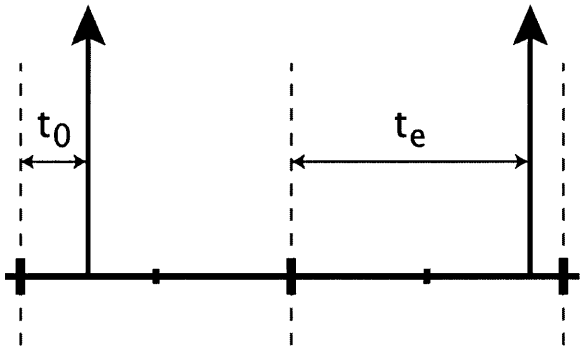


Figure 7-3: This bit sequence (presumably embedded in an entire byte) can be rejected as $t_e \neq t_0$ and $t_e \neq t_1$; the start bit sequence as shown in figure 7-2 defines those lengths, and the receiver can reject any spurious bits that do not match those values.

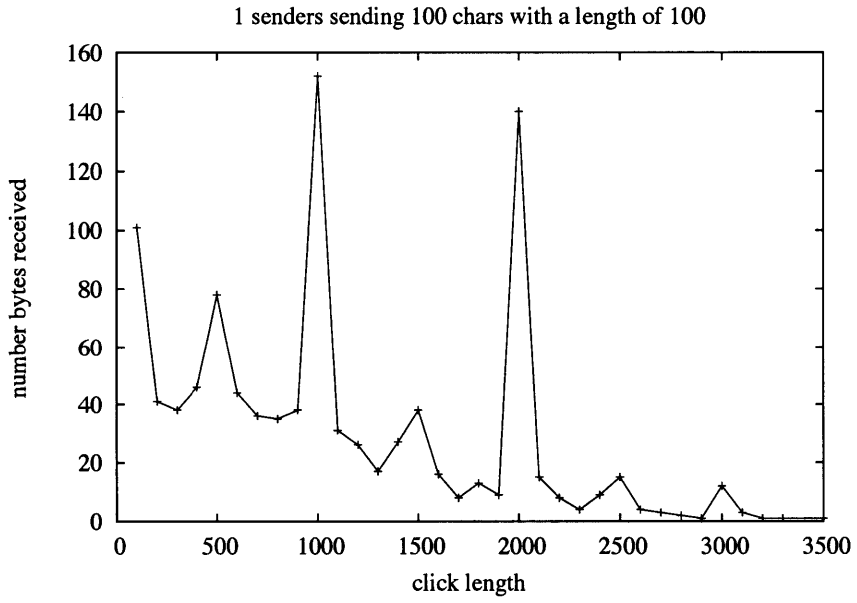


Figure 7-4: A plot of the number of bytes received at different click lengths when there is, in reality, only one sender transmitting 100 bytes with $t_c = 100$. There are "resonances" at $10t_c$ and another at $20t_c$.

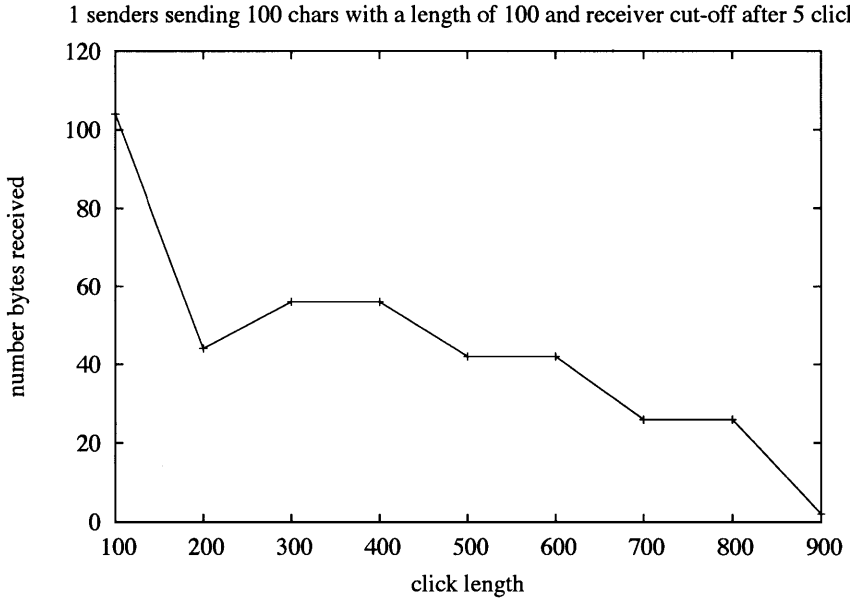


Figure 7-5: A plot of the number of bytes received at different click lengths when the receiver is rejecting any click interval that has more than 5 intervening clicks. There is only one transmitter who is transmitting 100 bytes with a $t_c = 100$.

sequence can be locked upon using a noise-locked loop or similar [Vig03] on the receiver end. An interesting property of this setup is that a transmitter can use a spreading sequence to transmit without consideration of whether the receiver is locking onto that spreading sequence, and using it to obtain coding-gain against the other extraneously decoded byte sequences; even if the receiver is not attempting to synchronize and use the spreading signal, the receiver can simply run the decoding algorithm and retrieve the information.

The receiver can also perform a significant amount of filtering on the clicks that it is receiving. One simple filter is to reject any click pairing that has more than a number of intervening clicks (see figure 7-5). In other words, no two subsequent clicks can be considered to belong to one byte if there are more than i other clicks occurring between them. Some knowledge is reflected in choosing the value of i, but when properly chosen, the number of extraneous other bytes decoded drops off significantly.

Another possibility is that a receiver can make two assumptions about the transmitter: a transmitter is not going to change its t_c in the middle of its stream, and that a transmitter is going to be sending a series of bytes in succession. One the receiver sees a series of bytes with the same t_c, it can start to make the assumption that that is the t_c of the transmitter. However, this assumption alone is not sufficient as that may confuse the receiver into believing that the transmitter is transmitting at a $t_c = 10t_c$ or $t_c = 20t_c$. Therefore, the receiver must also examine whether the bytes being received are overlapping as that would mean that the receiver is

transmitting two simultaneous bit sequences (which we assume it is not doing). Upon that examination, it is revealed that *all* the extraneous bytes in figure 7-4 are overlapping with other bytes – only those bytes with $t_c = 100$ can be ordered as a non-overlapping stream. Those bytes with a $t_c = 10t_c$ or a $t_c = 20t_c$ all have their byte sequences overlapping and occurring simultaneously. Also, a sender's byte sequences are usually sent in tight succession. This means that after the last click in the stop sequence, the first click in the start sequence can occur as early as t_i after that.

At the time of this writing, no one of these strategies have been settled upon as the "best".

7.3.2 Decoding in the presence of noise

This modulation scheme functions quite well in the presence of noise in the channel. A series of tests were performed with a single sender transmitting 100 bytes with a $t_c = 100$ in the presence of AWGN (see figure 7-6). Up to a noise floor of under -1 dB, this strategy functioned quite well rejecting noise that was spuriously detected as clicks. The number of extraneously decoded bytes were within the same number of bytes decoded in a noise free channel (figure 7-4).

After about -1 dB of noise, the frequency of the random clicks passes a threshold where they begin to occur often enough that the decoding algorithms will begin to find many byte sequences where there are none. Even filtering, as described above, only does clean up the received byte sequences (as the t_cs of the decoded bytes are spread almost as white noise) to an extent. Even if the receiver ignores click pairings with more than a given number of interleaving clicks, there are still a large number of clicks that can be formed under that limit. The only real possibility in such high noise situations for filtering is to more carefully design the hardware match filter (instead of using the naïve energy detector discussed above) to more discriminately fire only when an impulse is received. While the software algorithm will function, it will be prohibitively computationally expensive.

7.3.3 Decoding in the presence of multiple users

The one quite tantalizing aspect of impulse radio is the ability to cope with multiple users on the shared channel. Ultra-wide band treats all other users as "noise" when it is using its spreading sequence to time hop its positioned pulses; because the pseudo-random sequence is instructing the receiver to look at the channel at only very specific instants, any other impulse in the channel that is not occurring in that window is ignored. Internet 0, instead, relies on the self consistency of the byte sequences. In essence, it makes use of the fact that simultaneous byte sequences can be decoded.

In order to work properly, the transmitter needs to choose a random t_c on start-up and stay with that t_c value for as long as it is operating. It is very important that the t_c value be randomized, otherwise catastrophic collisions are possible in large networks of similar devices (see figure 7-7). Even a simple strategy of picking a mean t_c value, and then randomly settling on a t_c value within a deviation of that simplifies the amount

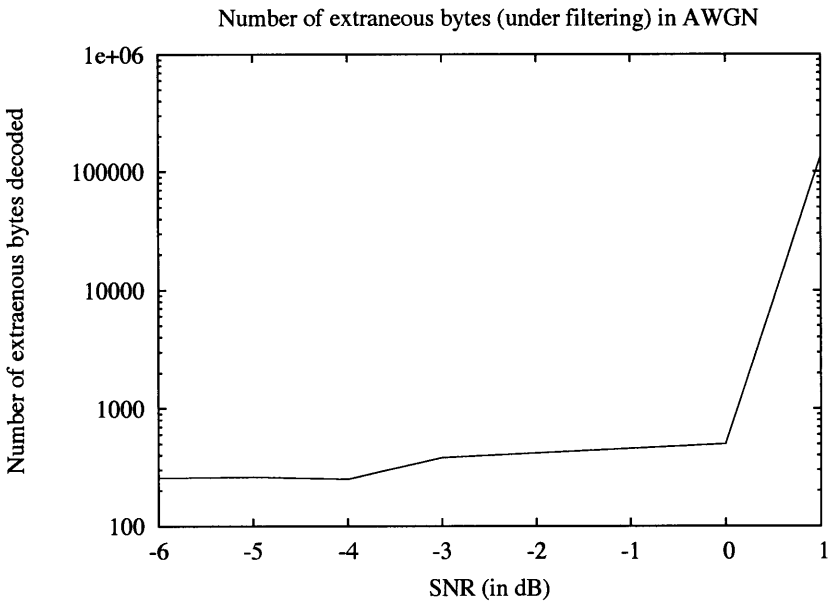


Figure 7-6: The number of decoded bytes with respect to a given level of AWGN on the channel as performed by the given decoding algorithm with filtering based on the number of intervening clicks. All tests were performed with one sender sending 100 bytes with a $t_c = 100$. The intervening click decoding strategy is fairly impervious to random noise below a certain level (save for the extraneous bit sequences that have not yet been filtered out) – however once a large number of random clicks is introduced (around a SNR of -1 dB), then the decoding algorithm begins to unwrap a large number of extraneous bytes even when fighting against intervening clicks.

of work that the receiver needs to perform (see figure 7-8). The receiver simply needs to run the decoding algorithm listed above, and then it can associate streams by assuming that each transmitter is not going to be changing its t_c value, and that each transmitter is going to be sending its bytes in succession. Using those two pieces of information, the receiver can string together the bytes into streams. It won't necessarily know which transmitter is transmitting which stream[2], however the receiver is able to separate out the streams.

The real draw to this type of information organization is that multiple access is now very simple to implement. Assuming that the transmitter is not transmitting at the same t_c as another (although it may still be possible, but computationally expensive, to separate out two transmitters at the same t_c), then the transmitters can blindly transmit onto a channel – there is no need for the transmitter to determine whether another is using the channel to schedule around it. The receiver will, in most cases, be able to decode the given click sequence.

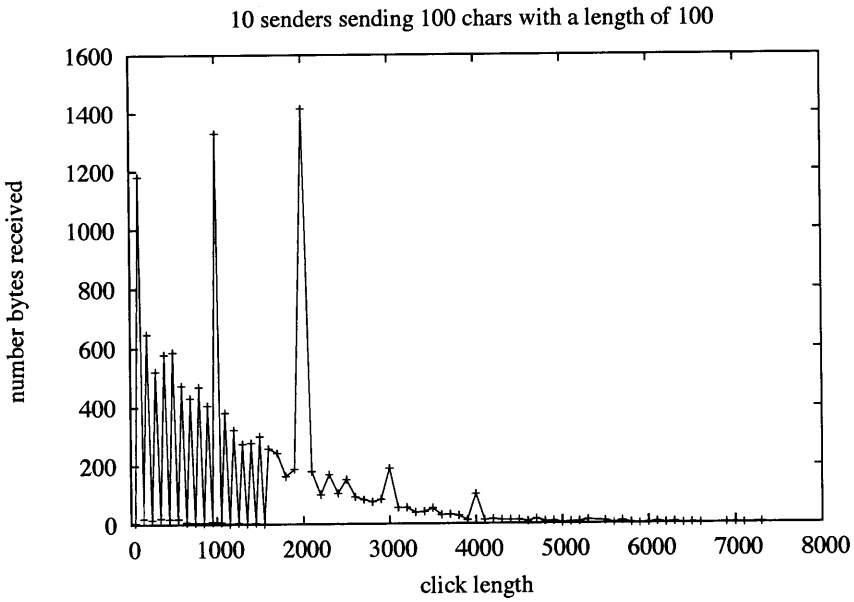
7.3.4 Breaking abstraction layers for decoding

Another very important heuristic that can be used for decoding is in the knowledge that this bit stream is carrying IP packets. Given that out-of-layer information, the receiver has the following more pieces of information and its disposal[3]:

1. The length of the packet,
2. the protocol of the packet,
3. and the checksum of the packet.

The length of the packet can tell the receiver how many more bytes to search for on this channel, while the protocol of the packet can deliver a few hints as to the structure of the data (specifically the next header) that we are going to be looking for. The checksum is very useful in that it can allow us to verify that the receiver is actually decoding the proper data. The checksum also becomes invaluable in the situation where noise causes the receiver to see a bit sequence that can decode to more than one byte[4] – the checksum can help filter out which one of those byte possibilities is the correct one.

[2]Unless it is encoded in the stream, such as the source address in an IP packet.
[3]Some of this information can only be trusted after the entire header has been received and the checksum verified, of course.
[4]This can happen if a noisy click occurs exactly at the location of a 0 bit when the transmitter is sending a 1 bit, or visa versa.



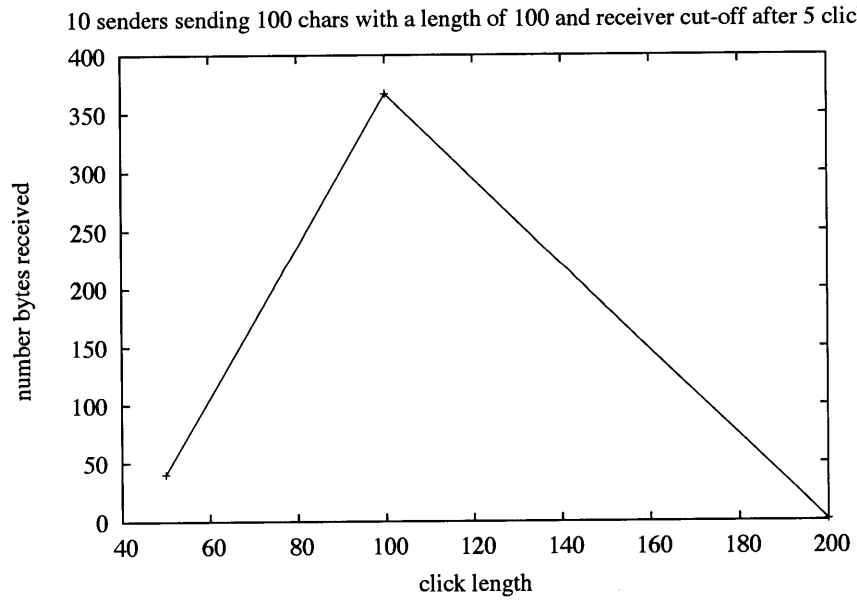
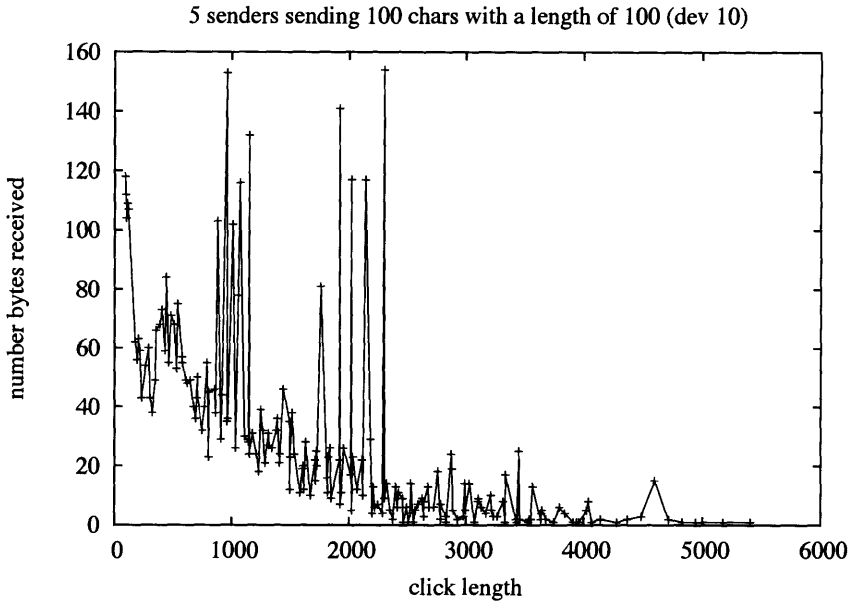


Figure 7-7: A plot of the number of bytes received (unfiltered and filtered) associated to the t_c value decoded at, with different click lengths in the presence of 10 senders who are all sending 10 bytes at a $t_c = 100$. Unfortunately, it becomes nearly impossible to form associations and streams keyed against senders with this plot as there is no way to distinguish between the senders (the data could be misinterpreted as a single sender at $t_c = 100$ instead of multiple senders).



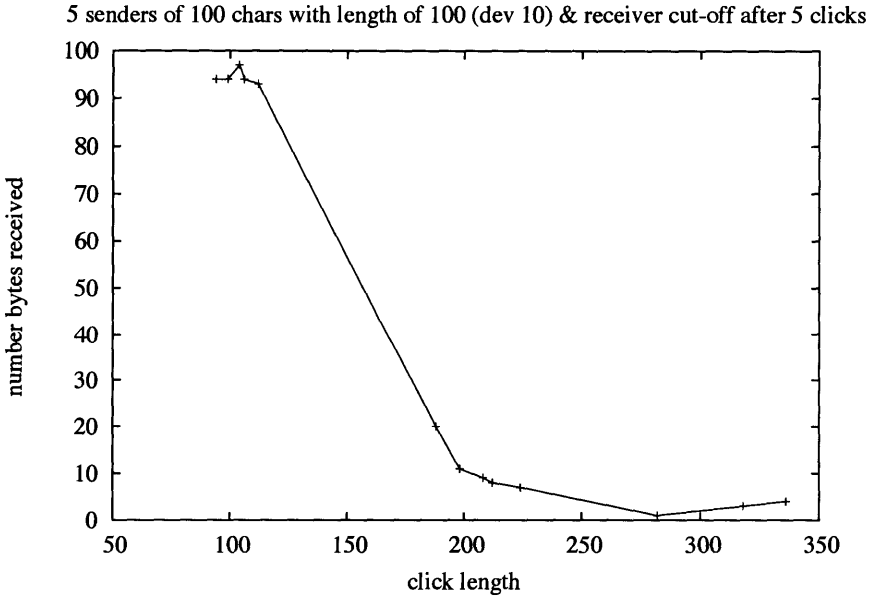


Figure 7-8: A plot of the number of bytes received (unfiltered and filtered) associated to the t_c value decoded at, with different click lengths in the presence of 5 senders who all have t_c values that are within a deviation of t_c and none of which are equal.

Chapter 8

Open Standards

Internet 0 is based around the same philosophical thought that the Internet is developed around: open up the standards that govern interactions. Allowing access to these specifications increases the ability for experimentation and innovation in ways that are not possible on closed systems. Access means shifting the market from building infrastructure, to building *better* infrastructure. Access means that everybody from students working on class projects, to massive corporations looking to develop unique solutions to problems can all work at the same base.

Mike O'Dell, former CTO of UUNET articulated this mindset very well in an e-mail message when discussing the spread of wireless community networks

> Biology will out-innovate the centralized planning "Department of Sanctified New Ideas" approach every time, if given half a chance. Once the law of large numbers kicks in, even in the presence of Sturgeon's Law ("90things happen that the progeny are at least interesting, and some are quite vigorous.
>
> "Nature works by conducting a zillion experiments in parallel and most of them die, but enough survive to keep the game interesting." (The "most of them die" part was, alas, overlooked by rabid investors in the latter 90s.)
>
> This is how the commercial Internet took off, and this remains the central value of that network ... You don't need permission to innovate.
>
> You get an idea and you can try it quickly. If it fails, fine – one more bad idea to not reinvent later. But if it works, it takes off and can spread like wildfire.

8.1 The failure of the American phone system

For contrast, consider the American cellular phone system. Until only recently, the US phone system had a myriad of competing infrastructures from Sprint's CDMA network to AT&T/Cingular's TDMA network to

Verizon's own CDMA network. The cellular phone consumer was left choosing a cellular phone provider simply by its network coverage map and not by the services provided. Simultaneously, these networks closed off access to the cellular phone manufacturer from the consumer – you had to purchase your cellular phone from the service provider you signed a contract with. A consumer could not simply purchase a phone from Nokia and then use it on the Sprint network. This was very different from the Europeans who settled fairly quickly onto the GSM standard for cellular phone connectivity thereby allowing Europeans to use their cellular phones anywhere on the continent. And, because the infrastructure was standardized and open, manufacturers could produce their own GSM phones and expect it to work anywhere. Consumers were not wedded to a series of phones that their service provider supports and they were free to purchase any GSM cellular phone and place their SIM card within it.

Arguably, this openness in the phone system has also lead to the fast rise of SMS technologies and the deploying of many more information services on the European networks. To launch a similar application on a network of a stateside provider requires much back and forth with marketers and executives at that network to convince them that the application is "worthwhile". This puts complete control in the network provider and stifles innovation. In the open European system, one simply purchases access to the network because, as a business case, if one provider does not give a consumer access, the consumer will simply go to another one. If a group of artists, students, or business people, wanted to create an information service that an end-user could interact with over a cellular phone, there is no company that they need to convince to deploy their application on, they simply need to purchase access to the network.

Although for any given link, the American technologies may have better sound or data quality, the closed interoperability of the networks has doomed it to failure. "Worse is better" when it comes to the deploying of these networks.

8.2 Internet 0 competitors

The lessons of openness all appear in the development of the Internet and are echoed in O'Dell's e-mail above. It is because of the generality and freedom of the Internet did a network that was once used primarily for login into remote machines evolve into one that electronic mail could be transferred over, to one where Gopher ran wild, and finally to one where the World Wide Web dominates in its usage. At no point did any one person determine that a particular way of organizing global information was a better way – people were free to experiment in parallel, and the more popular way would always win out. Because people simply negotiate access to the network, without having to defend their reason as to why they want to get access. And because the network specifications are open in the way that you interact with the network, did the Internet evolve the way it did with any random software and hardware developer being able to deploy their products and making the Internet a richer place.

In contrast to the open system, there are a few competitors to Internet 0 which are not as open: Echelon's

LonWorks, UPnP, and BACnet are two of them. Both of these has a few nuances that close out the system either in the engineering, or in the business domain.

8.2.1 LonWorks

LonWorks is one of the more predominant networks used for building control systems today. However, to develop systems using this control network requires membership to the Open Systems Alliance – a group of corporations lead by Echelon technologies, the main producer and designer of LonWorks technologies. However, development requires joining either the "Authorized Network Integrator" or "LonWorks Integrator" programs – both of which necessitates the purchase of a STARTUP package. Sadly, obtaining the title of either of these integrators does not necessarily mean that one is empowered to create devices that can be integrated into a network speaking LonTalk, but instead it means that the organization is recognized as being able to deploy a network and devices into it. In order to actually develop devices and software that interact with the network directly requires licensing the LonTalk protocol, as the Echelon corporation has a patent on the specification.

8.2.2 UPnP

UPnP is the universal plug-n-play standard that is meant to allow a device to automatically configure itself on a network. Obtaining the standards for UPnP (such as the Internet Gateway Device standard) is simple, however advertising the product as UPnP compliant requires registration into the UPnP Implementors Corporation at a $5000 per year fee. Therefore, while it is possible to create a UPnP compliant device, it is not allowed to advertise it to the world as UPnP compliant unless one joins the corporation. This leaves many smaller developers and experimenters in a catch-22.

8.2.3 BACNet

This third competitor, BACNet, is the most open of the three being discussed. BACNet is a global ISO standard that defines the networking protocol that is to be used between any two nodes in the network. Additionally, the standard is free for anybody to obtain and implement. However, there is no transparency into the organization for protocol development. If another feature is needed in the protocol, there is no standard way to add it. The benefit of Internet 0 is that only the lower layer protocol is defined, and it allows any other standard to be implemented at the higher levels (including BACNet, if desired).

8.3 Openness in Internet 0

It is the hope that by relying and championing completely open Internet standards, the development of inter-device inter-networks will be spurred on as anybody can implement it. Every niche in the networking will

be filled, or at least occupied, by somebody who is willing and interested to take charge – whether it be networking via drop-outs in LED illumination, to networking via short range pulses in the 2.4 GHz domain, to networking via the ground-water pipes in a home – all can be interoperable and all similarly functional.

None of what is said above should be needed to be articulated, however, it sometimes needs to be said to spur competition in the development of services, and not in the development of infrastructure.

Chapter 9

Conclusion

While all the previous themes do provide a methodology to create a device network that is interoperable with itself at the modulation layer and compatible with the global Internet on the protocol level – all of the presented are only possible because of careful observations and engineering. No one of those stages were rigorously designed and analyzed as they were only put together out from engineering experience. On other words, what is still lacking is a firm theoretical foundation on anything presented above.

There are questions as to whether the protocols used on the Internet are even the correct ones to be used in a device setting. More questions present themselves about delayering, and how it applies to engineering practice. And finally, there more open paths when considering peer-to-peer and decentralized systems.

9.1 Protocol correctness

Chapter 2 discussed the need for the use of Internet Protocols in inter-device networks as a way to maintain compatibility with the global Internet. The IP protocol, as well as the TCP one, were designed by a group of smart people who, engineering-wise, strongly felt that the protocol was correct. Only recently have groups begun to model the Internet and protocols [LAWD04] and this modeling has asked questions as to whether our current implementations are the optimal ones [PWDL04] [GG03]. The consensus is that TCP and IP are the correct solutions to networking, however a few optimizations can be made to TCP involving ACK transmission times for performances improvements [JWL04].

What is lacking, however, is an analysis of optimal, yet minimal, protocols that achieve the same goals of IP with a minimum number of states. The creation of the state diagram for the micro-IP stack (figure 2-1) shows an attempt from the standpoint of a software engineer, however can it be done better and be done smaller? Could IP be further optimized if one is assuming a global network of not only computationally powerful machines such as desktop computers and rack-mount servers, but also those of light switches and bulbs?

9.2 Delayering

The concept of working across software layer boundaries was mentioned in chapter 3. Metaphorically, one would like to consider a network specification to be a software program that a compiler then optimizes down. To do this, one needs to be able to represent the layers of a system individually, but then jointly optimize across them to hopefully create a smaller and more efficient system. Analysis such as this have already been performed to balance out the network and the physical layer in wireless networks [Chi05], however the more general question is whether to layer or not [Chi04] when designing multi-layer protocols.

The micro-TCP stack presented in chapter 2 (see figures 2-3 and 2-4), and the statement of breaking the abstraction layer to use IP information in decoding end-to-end modulated clicks (section 7.3.4) are two examples of creative engineering that has only been tested in simulation and not tested with mathematical rigor. Using joint estimation techniques may give us better insight on how to do that better, or how to bring in other axes of information.

9.3 Peer-to-peer message passing

Lastly, peer-to-peer systems as described in chapter 4 have been used in engineering practice as a way of harvesting more computation and storage [YGM01] [But02]. While some work is being done in creating compilers that can take directions for an entire network and produce software that runs on each individual node to collectively create the desired behavior [See02], most work simply has smart people who craft the protocols and split up the computation manually. The product-sum algorithm [KFL01] provides some insight on how to distribute probabilistic computations amongst communicating nodes, and work in this field may involve the distribution of routing algorithms or other control algorithms throughout the network.

9.4 Final remarks

As said above, Internet 0 is simply a framework for reasoning about and designing interoperable device networks. Most device networks do embody subsets of the seven themes presented, but no framework except I0 involves all of them. However, most device networks have not yet attempted to scale to the size of the Internet – there is much anecdotal and theoretical proof that has scaled properly, it seems irresponsible not to use it, or at least learn from it. I0 provides a collection of thoughts that if implemented correctly could lead to the rapid development of devices that are capable of not only communicating across a PCB board, but also across the world.

Bibliography

[AGK99] George Apostolopoulos, Roch Guerin, and Sanjay Kamat. Implementation and performance measurements of qos routing extensions to OSPF. In *INFOCOM (2)*, pages 680–688, 1999.

[BP] Ben Bauer and Andrew S. Patrick. A human factors extension to the seven-layer osi reference model.

[But02] William Butera. *Programming a Paintable Computer*. PhD thesis, Massachusetts Institute of Technology, 2002.

[CB86] R. W. Callon and H. W. Braun. RFC 986: Guidelines for the use of Internet-IP addresses in the ISO Connectionless-Mode Network Protocol, June 1986. Obsoleted by RFC1069 [CB89]. Status: UNKNOWN.

[CB89] R. W. Callon and H. W. Braun. RFC 1069: Guidelines for the use of Internet-IP addresses in the ISO Connectionless-Mode Network Protocol, February 1989. Obsoletes RFC0986 [CB86]. Status: UNKNOWN.

[Chi04] M. Chiang. To layer or not to layer: balancing transport and physical layers in wireless multihop networks. In *Proceedings of IEEE INFOCOM*, 2004.

[Chi05] M. Chiang. Balancing transport and physical layers in wireless multihop networks: Jointly optimal congestion control and power control, 2005.

[Coh81] Danny Cohen. On Holy Wars and a plea for peace. *j-COMPUTER*, 14(10):48–54, October 1981. This is an entertaining account of the Big-Endian and Little-Endian problems of bit and byte ordering.

[DC85] S. E. Deering and D. R. Cheriton. RFC 966: Host groups: A multicast extension to the Internet Protocol, December 1985. Obsoleted by RFC0988 [Dee86]. Status: UNKNOWN.

[Dee86] S. E. Deering. RFC 988: Host extensions for IP multicasting, July 1986. Obsoleted by RFC1054, RFC1112 [Dee88, Dee89]. Obsoletes RFC0966 [DC85]. Status: UNKNOWN.

[Dee88] S. E. Deering. RFC 1054: Host extensions for IP multicasting, May 1988. Obsoleted by RFC1112 [Dee89]. Obsoletes RFC0988 [Dee86]. Status: UNKNOWN.

[Dee89] S. E. Deering. RFC 1112: Host extensions for IP multicasting, August 1989. Obsoletes RFC0988, RFC1054 [Dee86, Dee88]. Updated by RFC2236 [Fen97]. Status: STANDARD.

[DNP99] M. Degermark, B. Nordgren, and S. Pink. RFC 2507: IP header compression, February 1999. Status: PROPOSED STANDARD.

[EF94] K. Egevang and P. Francis. RFC 1631: The IP network address translator (NAT), May 1994. Status: INFORMATIONAL.

[Fen97] W. Fenner. RFC 2236: Internet Group Management Protocol, version 2, November 1997. Updates RFC1112 [Dee89]. Status: PROPOSED STANDARD.

[FLYV92] V. Fuller, T. Li, J. Yu, and K. Varadhan. RFC 1338: Supernetting: an address assignment and aggregation strategy, June 1992. Obsoleted by RFC1519 [FLYV93]. Status: INFORMATIONAL.

[FLYV93] V. Fuller, T. Li, J. Yu, and K. Varadhan. RFC 1519: Classless inter-domain routing (CIDR): an address assignment and aggregation strategy, September 1993. Obsoletes RFC1338 [FLYV92]. Status: PROPOSED STANDARD.

[GC93] Paulo Guedes and Miguel Castro. Distributed Shared Object Memory. In *Proc. 4th Wshop. on Workstation Operating Systems (WWOS-IV)*, Napa, CA (USA), 1993. IEEE Computer Society Press.

[GG03] Y. Gu and R. Grossman. End-to-end congestion control for high performance data transfer, 2003.

[JWL04] Cheng Jin, David Wei, and Steven Low. Fast tcp: motivation, architecture, algorithms, performance. In *IEEE Infocom*, mar 2004.

[KFL01] Kschischang, Frey, and Loeliger. Factor graphs and the sum-product algorithm. *IEEETIT: IEEE Transactions on Information Theory*, 47, 2001.

[LAWD04] L. Li, D. Alderson, W. Willinger, and J. Doyle. A first principles approach to understanding the internet's router-level toplogy. In *Proceedings of the ACM SIGCOMM Confernce on Network Architectures and Protocols*, 2004.

[LMLL03] D. Laney, G. Maggio, F. Lehmann, and L. Larson. A pseudo-random time hopping scheme for uwb impulse radio exploiting bit-interleaved coded moduatlion. In *Proceedings of the 2003 International Workshop on Ultra Wideband Systems*, 2003.

[LZ75] Barbara Liskov and Stephen Zilles. Specification techniques for data abstractions. In *Proceedings of the international conference on Reliable software*, pages 72–87, 1975.

[Mil94] David L. Mills. Internet time synchronization: The network time protocol. In *Zhonghua Yang and T. Anthony Marsland (Eds.), Global States and Time in Distributed Systems, IEEE Computer Society Press*. 1994.

[Min99] N. Minar. A survey of the ntp network, 1999.

[MKL+] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. Peer-to-peer computing.

[Mog84a] J. C. Mogul. RFC 919: Broadcasting Internet datagrams, October 1984. Status: STANDARD.

[Mog84b] J. C. Mogul. RFC 922: Broadcasting Internet datagrams in the presence of subnets, October 1984. Status: STANDARD.

[MP85] J. C. Mogul and J. Postel. RFC 950: Internet Standard Subnetting Procedure, August 1985. Updates RFC0792 [Pos81c]. Status: STANDARD.

[MS93] Andrew Myles and David Skellern. Comparing four IP based mobile host protocols. *Computer Networks and ISDN Systems*, 26(3):349–355, 1993.

[MVS01] David Moore, Geoffrey M. Voelker, and Stefan Savage. Inferring internet Denial-of-Service activity. In *Proceedings of the 2001 USENIX Security Symposium*, pages 9–22, 2001.

[Pap03] Ravi Pappu. *Physical One-Way Functions*. PhD thesis, Massachusetts Institute of Technology, 2003.

[Pos80a] J. Postel. RFC 760: DoD standard Internet Protocol, January 1980. Obsoleted by RFC0791, RFC0777 [Pos81b, Pos81a]. Status: UNKNOWN. Not online.

[Pos80b] J. Postel. RFC 768: User datagram protocol, August 1980. Status: STANDARD.

[Pos81a] J. Postel. RFC 777: Internet Control Message Protocol, April 1981. Obsoleted by RFC0792 [Pos81c]. Obsoletes RFC0760 [Pos80a]. Status: UNKNOWN. Not online.

[Pos81b] J. Postel. RFC 791: Internet Protocol, September 1981. Obsoletes RFC0760 [Pos80a]. Status: STANDARD.

[Pos81c] J. Postel. RFC 792: Internet Control Message Protocol, September 1981. Obsoletes RFC0777 [Pos81a]. Updated by RFC0950 [MP85]. Status: STANDARD.

[Pos81d] J. Postel. RFC 793: Transmission control protocol, September 1981. Status: STANDARD.

[PWDL04] F. Paganini, Z. Wang, J. Doyle, and S. Low. Congestion control for high performance, stability, and fairness in general networks, 2004.

[Ray98] Eric Raymond. Goodbye, 'free software': hello, 'open source', February 1998.

[RP94] J. Reynolds and J. Postel. RFC 1700: ASSIGNED NUMBERS, October 1994. Status: STANDARD.

[SAA03] David Stirling and Firas Al-Ali. Zero configuration networking. *Crossroads*, 9(4):19–23, 2003.

[SD94] Sandeep Sibal and Antonio DeSimone. Controlling alternate routing in general-mesh packet flow networks. In *SIGCOMM*, pages 168–179, 1994.

[See02] Devasenapathi Seetharamakrishnan. C@t: A language for programming massively distributed emebedded systems. Master's thesis, Massachusetts Institute of Technology, 2002.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Francs Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.

[SRC84] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design, 1984.

[SW02] Subhabrata Sen and Jia Wong. Analyzing peer-to-peer traffic across large networks. In *Second Annual ACM Internet Measurement Workshop*, November 2002.

[Vig03] Benjamin Vigoda. *Analog Logic: Continuous-Time Analog Circuits for Statistical Signal Processing*. PhD thesis, Massachusetts Institute of Technology, 2003.

[WHH+92] Carl A. Waldspurger, Tad Hogg, Bernardo A. Huberman, Jeffrey O. Kephart, and W. Scott Stornetta. Spawn: A distributed computational economy. *Software Engineering*, 18(2):103–117, 1992.

[WS98] M. Win and R. Scholtz. Impulse radio: how it works, 1998.

[WS00] M. Win and R. Scholtz. Ultra-wide bandwidth time-hopping spread-spectrum impulse radio for wireless multiple-access communications, 2000.

[YGM01] Beverly Yang and Hector Garcia-Molina. Comparing hybrid peer-to-peer systems. In *The VLDB Journal*, pages 561–570, sep 2001.

[Zis97] A. Zisman. A methodology to assist with a distributed information discovery process for autonomous databases, 1997.