TCP/IP enabled legOS

Student research project

University of applied sciences Hamburg

March 3, 2002

Olaf Christ
christ_o@gmx.de

Abstract

In recent years, the interest for connecting small devices such as sensors or embedded systems into an existing network infrastructure (such as the global Internet) has steadily increased.
Such devices often have very limited CPU and memory resources and may not be able to run an instance of the TCP/IP protocol suite.
This document describes how to use the uIP TCP/IP stack, written by Adam Dunkels, on LEGO's RCX Mindstorm platform, a very small device / toy powered by a h8300 Hitachi microcontroller with very limited RAM (32K) and processing power. Still, it is powerful enough to run alternative operating systems such as LegOS or even a tiny Java VM as a replacement of the original firmware. The advanced user is usually not satisfied with the original firmware due to its extremely limited capabilities in terms of executing complex code and the very limited number of variables.
The uIP TCP/IP stack is intended for embedded systems running on low-end 8 or 16-bit microcontrollers. The code size of uIP is an order of a magnitude smaller than similar generic TCP/IP stacks available today.

The uIP code and an up-to-date version of the uIP documentation can be downloaded from the uIP homepage maintained by Adam Dunkels.

Preface

This work has been carried out as a student research project at the University of applied sciences Hamburg in Hamburg, Germany.
The proposal was given to me by my supervisor Prof. Kai v. Luck.
I agreed, as the project involves both coding and research: I already had developed an interest for programming very small embedded systems during my work on my own robot-contestant for the annual robot contest at the University of applied sciences Hamburg.
This work has not only given me insights into the low-level function of the TCP/IP protocols, but also into the problems occuring within infrared wireless networks.

This report has been written using MS-Word.
The code has been written using the VI editor and compiled with the GCC compiler, which is acting as a cross-compiler for the H8300 processor. The code has been run on the Linux operating system.

Olaf Christ
Hamburg, 31 December 2001

Acknowledgments

First and foremost, I'd like to thank Adam Dunkels for writing uIP and providing me with extensive support. I'd also like to thank Martin Cornelius for providing me with crucial information about the LegOS kernel, especially about the LNP-protocol and the kernel's lowlevel details. Ross Crawford and Paolo Masetti deserve credits for additional help.
Thanks also must go to Christian Schmelzer for helping me with English and doing the proofreading.

Finally, I'd like to thank Markus L. Noga for starting LegOS, one of the most exiting open source projects today.

Introduction

In recent years, the interest for connecting both computers and computer supported devices to wireless networks has steadily increased. Nowadays, embedded systems such as the RCX or MIT's Handyboard are affordable at very litte cost. Computers are becoming more seamlessly integrated with everyday equipment and prices are dropping. At the same time wireless networking technologies, such as Bluetooth [HNI + 98] and IEEE 802.11 WaveLAN [BIG + 97], are emerging.
Even though this paper does not actually deal with those protocols, it gives a good insight into the obstacles occuring with wireless networks and the integration of tcp/ip into small devices.
Connecting small devices such as sensors or robots to an existing network infrastructure plus the possibility of monitoring them from anywhere gives a glimpse of exciting new products to come.

The Internet technology has proven itself flexible enough to incorporate the changing network environments of the past few decades. While originally developed for low speed networks such as the ARPANET, the Internet technology today runs over a large spectrum of link technologies with vastly different characteristics in terms of bandwidth and bit error rate.
It is highly favorably to use the existing Internet technology in wireless networks of tomorrow since a large amount of todays internet applications rely on that technology. Also, the large connectivity of the global Internet is a strong motivation

Since small devices such as sensors or robots are often required to be physically small and inexpensive, an implementation of the Internet protocols will have to deal with both having little computing resources and limited memory.
How to incorporate a tcp/ip protocol (stack) into such small devices is the topic of this student research project using one the worlds tiniest stacks available.

1.1 Goals

There is only one goal:
- setting up a connection oriented wirelss network connection between the PC and the Mindstorm using a standard network protocol (tcp/ip)

Because of the very limited RAM the tcp/ip implementation and the accomanying code should still leave enough free RAM for own usercode.
The tcp/ip stack uIP is sufficiently small in terms of code size and resource demands to be used in a small system such as the RCX.
As a concrete example a wireless connection between the PC and the RCX will be established using ping to evaluate the quality of the connection in terms of packet loss and roundtrip times.
The connection-orientation will be achived using the uIP tcp/ip stack by Adam Dunkels and the given integrity layer of the LNP protocol, a layer currently unused by the LNP-protocol.

1.2 Approach

The approach to integrate the stack into the LegOS kernel, is to use the, unused, integrity layer as a transport media and the lnpd on the pc side, by making adequate changes to the kernel such as removing the sound support to gain extra RAM.
Writing a gateway is needed to establish the communication between the two worlds, the RCX and the PC.

1.2 Methodology and limitations

The research in this work has been conducted in an experimental fashion.
After an initial idea is sprung, informal reasoning around the idea gives insights into its soundness. Those ideas that are found to be sound are then implemented and tested to see if they should be further pursued or discarded. If an idea should be further pursued it is refned and further discussed.

The goal was achieved using the vtun-device driver and the lnpd to route and tunnel each packet sent between the PC and the RCX (the LNPD, to be exact) to make them available to e.g. 192.168.0.1.
Because of the very slow speed you cant expect very high performance. Packet losses will dramatically increase if a range of 3 meters is exeeded.
No formal test suites have been used.
Testing was done manually by using small programs to test every aspect of the connection indvidually, e.g. sending data back and forth between the PC and the RCX.

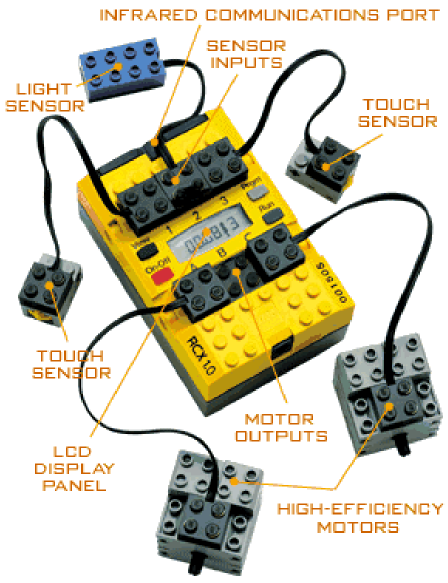
1.4 Structure

The report is organized as follows.
Chapter 2 provides a background of the RCX Internals, LNP and Wireless Networks.
Chapter 3 describes the uIP stack and the usage of TCP/IP in wireless networks.
Chapter 4 describes how LNP and TCP work together and how lnpd and a socket are used as a TCP/IP-lnpd-Gateway.
Chapter 5 descibes alternative approaches.
Chapter 5 descibes alternative approaches.
Chapter 6 gives explains why performance will degrade.
Chapter 7 an example of use
Chapter 8 summarizes the work
Chapter 9 descibes what could be done in future projects.

Appendix A gives a detailed introduction to TCP/IP
Appendix B contains some code examples
Appendix C contains a glossary.

2 Background

2.1 The RCX-Hardware



I am not going to describe the Harware of the RCX in detail. Kekoa proudfoot has already done this.
You can view his very detailed and quite large description of the RCX at his homepage
I am only giving you a quick glimpse at the most important facts you should always have in mind when writing any code fo the RCX. (regardless of what kind of OS you are using)
The core of the RCX is build upon the Hitachi H8 microcontroller.with only 32k of external RAM.
The microconroller of the RCX is used to control, three motors, three sensors and an infrared tranceiver connected to the serial communications port.
An on-chip, 16K ROM contains a driver that is run when the RCX is first powered up.
This driver is extended by downloading 16K of firmware to the RCX.

Communication between the RCX and an infrared tranceiver, connected to one of the PC's serial ports is achieved using a serial line IR protocol.

2.2 The LNP-protocol

The LNP-protocol is a quite simple UDP-like protocol, specifically designed for LegOS.
LNP currently offers two types of packets.
These two types are called the Integrity packet and the addressing packet.

The Integrity packet
The Integrity packet does not carry any addressing information, its could be considered as the LNP broadcast mechanism.

F0	LEN	IDATA	CHK

F0	: identifies an integrity packet
LEN	: length of IDATA section, 0 to 255
IDATA	: payload data
CHK	: checksum

Because LEN is one byte and the maximum length of the IDATA is 255, the overall maximum length of an LNP packet is 258 bytes.

The addressing packet.
The addressing packet is an integrity packet, encapsulating both payload data and addressing information in the IDATA section:

F1	LEN	DEST	SRC	ÂDATA	CHK

$\longleftarrow$ IDATA $\longrightarrow$

F1	: identifies an addressing packet
LEN	: length of IDATA section (1 to 255 bytes)
DEST	: destination address
SRC	: source addressADATA : payload data (1 to 253 bytes)

LNP-addressing packets are quite similar to UDP: they are not guaranteed to arrive, but if they arrive, they will contain no errors. An addressing packet is always sent to a specific port on a specific host. If the host is running a server on this port (an addressing-handler in LNP-idiom), and the packet arrives without errors, the packet is passed to the handler, otherwise (no handler or errors), the packet is discarded.

DEST and SRC carry the destination address, the originating address of the packet, respectively. Both are 1 byte, which is split into a host and a port section. The macro CONF_LNP_HOSTMASK (config.h) determines the individual bits of this byte carrying eitther the the host address information or port information, thus making it possible to configure the addressing scheme to your needs. E.g. we have got only 2 hosts (a PC and a RCX), but we want to have 128 unique ports on each host, you could set CONF_LNP_HOSTMASK to 0x80. Actually, it is not neccessary to use bit-schemes like 10000000, 11000000, 11100000, ... for CONF_LNP_HOSTMASK, but it is good wise to do so ! Otherwise precious RAM will be wasted inside LNP.

In most stituations the default CONF_LNP_HOSTMASK of 0xF0 is a good choice. Using this setup, the upper 4 bits of DEST and SRC are the host address, leaving the lower 4 bits for the port address. This allows you to assign 16 different hostaddresses, and 16 different ports on each host.

Setting up the source host address for the RCX
The destination address (HOST/PORT) is specified as a parameter of lnp_addressing_write(), specifying the destination for every packet sent in by function.
The hostaddress is defined in CONF_LNP_HOSTADDR in conf.h in the /legOS/boot, /legOS/util/dll.src repectively.

2.3 The LNP-deamon

Also called lnpd, a Linux deamon, written by Martin Cornelius, makes it easy to comminicate with the RCX, running LegOS.
The lnpd is not necessarily needed to work with the RCX and legOS.
The lnpd is only needed if we want to set up a communication between the usercode on the RCX and code on the PC.
This widely used deamon is not a part of the LegOS distribution but can be downloaded seperately from various sites.
Using the lnpd is rather simple, everything to do is linking the lnpd to the application to use the functions.
A very good example comes with the lnpd distribution itself.
The distribution contains all the sourcecode to succesfully build the deamon on a linux platform.
The examples that come with the distribution explain how to use both types of packets, integrity and addressing, to send data between the PC and the RCX.

2.4 The LegOS-Kernel

Because there already is a very detailed introduction to the LegOS Kernel written by Stig Nielsson.[NIE2000] and this report focuses on the networking capabilities of the RCX / LegOS system, i will only give you a very rough overview to the LegOS kernel so you will understand what happens if you download code or simply send data to the RCX.

I will be doing this by explaining all the stages an LNP-packet has to go through before its data is written into the RCX's memory.

Prerequisites:
I assume the RCX' is already powered up and the frimware has already been replaced by the LegOS kernel.
I also assume we re running the lnpd on our LINUX.with the tranceiver connected to the serial port.
I also assume that no patches have been applied to the LegOS kernel, because the available patches usually add code to the standard kernel and we may therefore run into a memory shortage.
I assume that we are runnging at least LegOS version 0.2.4.

Before data is to be sent over the IR connection it has to be passed to the lnpd.
The only things we have to do is to set up either an adressing handler or the so called integrity handler.
The only difference between these two handlers are that the addressing handler uses LNP-Packets augmented with the source and the destination address.
Please note, that LNP is a UDP-like protocol with all the consequences.
A UDP-like protocol does not guarantee that a packet reaches its destination!
This is very important if we need to know that every single packet has reached its destination e.g. if we are sending code to the RCX.
But if it does, makes it very unlikely that the data encapsulated in the packet contains errors.

The adressing handler is used to send so called adressing packets, packets, following the adressing scheme, over the network.
A connection using integrity packets can be seen as LNP's broadcast funcionality, whereas a connection based on adressing packets can be seen as LNP's way to establish a **connectionless** point to point connection.

It doesn't really matter what kind of packets we are using, especially if only one pc and only one RCX are involved.
For further information about LNP take a look at *2.2 The LNP-protocol* and *2.3 The LNP-deamon*.[COR2000]

On the PC:

The path of our data starts by calling something like

lnp_addressing_write(const unsigned char *data,unsigned char length, unsigned char dest,unsigned char srcport)

This function basically builds a LNP packet out of the passed parameters, following the LNP adressing scheme and passes it to the function lnp_logical_write (unsigned char *data, int length).
lnp_logical_write(), then, does all the necessary work by directly communicating with the socket to transmit the packet over the serial line / IR connection.

Assuming the packet has reached its destination, the RCX, we are ready to continue following the way of our packet to the RCX's memory.

On the RCX:

Packet transmission on the RCX is done interrupt driven.(for indepth information about the interrupts and their addresses, please refer to Kekoa Proudfoot's excellent website)

All the work is done by the functions in **lnp-logical.c,** building a statemachine that does the receivng and sending of bytes, recognizing and decoding LNP-Frames and checking each frame for validity using a checksum algorithm.

One of the most important funtions is
static void rx_handler(void), that acts as the byte received interrupt handler

This very important handler not only takes care of packet transmission, it also performs the collison detection.

Each network node (RCX or a pc running e.g. the lnpd) basically has two states: **tx-state==TX_ACTIVE** (sendin) or **tx-state<TX_ACTIVE** (receiving).

Receiving single bytes is done interrupt driven in rx-handler().
When receiving (**tx-state<TX_ACTIVE)** bytes are passed to lnp_integrity_byte().
In this state of the statemachine LNP-Frames are recognised and checked for validity.
Valid, well formed, LNP-fames are then passed to the lnp_receive_packet() that eventually passes it, if host-id AND port are valid, to the installed hander or otherwise ignores the frame.
Note: you have to set up your own handler (addressing or integrity) to tell legOS where to put the data.
Good examples how to set up handlers are the lnptest.c for the RCX (part of the lnpd distribution, to be found in /lnpd+liblnpd/rcx) or program.c, which is "nothing more" than an addressing handler, although a very important one.
This is also called the program handler and is besides other things, responsible for program download.
During program download, packets, the ones sent by lx, a program to download usercode to the RCX, are passed to the program handler.
The handler then parses the packet for specific information and eventually copies the data into the RCX's memory.

The hostaddress is defined in CONF_LNP_HOSTADDR in conf.h in the /legOS/boot, /legOS/util/dll.src repectively.

Sending data from the RCX to the PC is done almost excactly in the same way as it is done from the PC to the RCX.

The collisision detection algorithm

The collision detection algorithm checks whether or not own bytes have been received., which obiously is a bad thing, because collisions mutilate packets, that have to be resent and thereore reduce throughput.

If a collision has been detected, the function txend_handler() is called to abort the transmission.
To prevent two active nodes from blocking eachother (deadlock) a standard algorithm is used (CSMA/CD)

After receiving an arbitrary byte, sending bytes is forbidden for period defined in LNP_BYTE_SAFE.
The RCX checks if it is safe to send another byte (carrier sense).
After successfilly sending a frame, sending is forbidden again for a preriod defined in LNP_WAIT_TXOK.
This gives other nodes, also ready to send, the chance to get rid of their frames.
If a collision occurs, sending is formbidden for an even longer time `allow_tx`, which, is in fact the time plus a random value.
This is very important, because it prevents nodes, that are ready to send, from blocking eachother forever (deadlock).

2.5 Reliability and performance of LNP

Because LNP is UDP-like ,LNP-connections are not reliable. E.g. there is no resend. The only thing we can rely on is, that received packets are checked for validity. We *can* be quite sure that LNP packets contain correct data.

Using UDP-like protocols in general is not a bad thing.
The performance offered by UDP is usally very good and it can be implemented in a very few bytes, which is important when using systems with only 32K of RAM or even less.
But, nothing comes without a price, and using UDP is not a good choice if reliability is inherently neccessery.

For example when downloading a program to the RCX, every single byte must be transmitted, error free !
To assure this, the program handler sends back a handshake packet to the PC. The PC waits for this handshake before sending the next packet.
Unfortunaly,this has only been implemented for program downloading and is not done in lnp.c, where the LNP-protocol is implemted. And it is not a standard protocol.

Another example is, if you want to remote control the RCX. Imagine the RCX is equipped with a litte fan and an infrared sensor. The RCX is build into a small robot. The robot should look for a candle, drive to it an check whether or not it is lit. The robot then should use the fan to blow out the candle. Lets imagine we have more than one robot and the PC tells the closest robot to blow out the candle.
If one robot successfully extinguishes the candle, the robot can send an acknowledge to the PC and the PC can tell other robots not to come to the candle, because it already has been extiguished.
The nice thing about a reliable connections is, that the PC and the RCX can both be sure that their message has reached its destination.
In this example, knowing helps the PC to plan the next moves, which is important in multi agent applications such as e.g. robot soccer.

UDP *is* a great choice if you want to transmit e.g. audio data.and performance is much more important than correctness.
Nobody, usually, will ever notice if a couple of bytes in an audiostream are wrong.
Also games like QUAKE use UDP, because it does not really matter if a couple of bytes are wrong; nobody will ever notice.them.

Things like collision detection and preventing deadlocks are ususally taken care of by the network card itself.

LNP has been implemented the way it is for four reasons, codesize, the fact, that UDP is usually sufficient and performs well in a wireless IR network and last but not least, because implementing TCP/IP in a very few bytes is an art itself and there are only very few stacks available today Btw. the uIP- stack was originally written for this project by Adam Dunkels. It’s maybe the worlds smallest TCP/IP-stack and most hardware independent stack written in pure C.

***Note**: a reliable connection, in general, is important if every byte is important.UDP usually offers better performance.*

2.6 Wireless networks using IR

The RCX uses IR to transmit its data to the tranceiver and vice versa.
Therefore it makes sense to say something about wireless networks using IR.
These problems usually affect the throughput of protocols such as tcp/ip which is used on the RCX.

Transmitting data to the RCX is not without any problems, because packet losses occur quite often.
The explanation is quite simple. The RCX's IR system is build upon cheap components and the transmitting power is rather low. Nevertheless, the maximum range where packet losses are still acceptable is about 3 meters. You also have to make sure, that the light in the room is not too bright.

In general, in Wireless LAN's using IR, packet losses may occur quite often, so the classical explanation (see TCP/IP and wireless networks) does fit here.
Infrared (IR) systems use very high frequencies that are just below visible light in the electromagnetic spectrum to carry data. Like light, infra red rays cannot penetrate opaque objects. They are either directed (line-of-sight) or diffuse technology.
Inexpensive directed systems, like the one built into the RCX, provide a very limited range of just about three feet and are typically used for PANs but occasionally are used in specific WLAN applications.
High performance directed IR is impractical for mobile users and is therefore used only to implement fixed subnetworks.
Diffuse (or reflective) IR WLAN systems do not require line-of-sight, but cells are limited to individual rooms.

All the above should be kept in mind to successfully set up an experimental setup (see Example of use)

3 TCP/IP

3.1 The uIP TCP/IP-stack

The uiP TCP/IP stack is specifically designed for very small systems such as the RCX by Adam Dunkels.
The stack features are:

Very small code size.
Very low RAM-usage, configurable at compile time.
Very tight stack usage.
IP, ICMP (ping) and TCP protocols.
Configurable amount of concurrently active TCP connections.
Configurable amount of passively listening (server) TCP connections.
Well commented sourcecode- nearly every other code line is a comment.
Up-to-date documentation available at http://dunkels.com/adam/uip/documentation.html
Includes a tiny web (HTTP) server with simple scripting abilities.
uIP is free.

RAM usage:
uIP uses 23 bytes of RAM for each TCP connection and 2 bytes of RAM for each listening TCP port. Other than that, uIP uses 9 bytes of RAM. A configuration with at most 8 active TCP connections, one listening port and a packet buffer that can hold 96 bytes will use 8 * 23 + 2 + 96 + 9 = 291 bytes of RAM.

The compiled code size may vary, but is always only a few kilobytes.

The uIP stack is proven to succesfully run on a wide range of systems.(see Adam Dunkels homepage for details)

To achieve such small codesize and portablity, unorthodox methods are used.
The stack uses the goto-statement to eliminate stackusage, because in systems like the RCX its very important to use as litte of RAM as possible.
Because the system uses many gotos there are no real layers.
The stack calls the user code itself to keep the overhead and RAM usage as small as possible, therefore the usercode and the stack are tied up very closely.
The stack also uses no interrupts and no system timer, wich makes it architectural neutral.

Because the code of the stack is so small it is yet quite good to understand.

3.2 TCP/IP and wireless network

The TCP/IP stack provides two very different transport mechanisms, TCP and UDP. UDP is minimal, with only header encapsulation. On the other hand, TCP offers all the features expected from a transport protocol, with flow control and end to end reliability. The TCP protocol has been optimised to deliver excellent throughput in a wide range of configurations. Comparing the performance of UDP and TCP on some Wireless LANs, it's not uncommon to see that UDP offers 25 % more user throughput than unidirectional TCP. The classical explanation is that the radio losses are seen as congestion by TCP, and therefore TCP drastically reduces the throughput. However, it has been tested that in Wireless LAN's that include stop and go retransmissions, the UDP test shows that 100 % of the packets are received and that they are delivered strictly in sequence, so this explanation is not neccessarily valid. The second explanation is the overhead of the TCP ack packets. Those packets don't carry any useful payload, and moreover they are small, so subject to a relatively high overhead due to protocol headers, that have no special meaning in an TCP ack packet. This explains most of the difference, but there is still some additional overhead unaccounted for. This additional overhead for TCP is in fact due to collisions between the TCP data packets and TCP ack packets over the medium. As opposed to TCP, UDP is unidirectional, so there are no such collisions

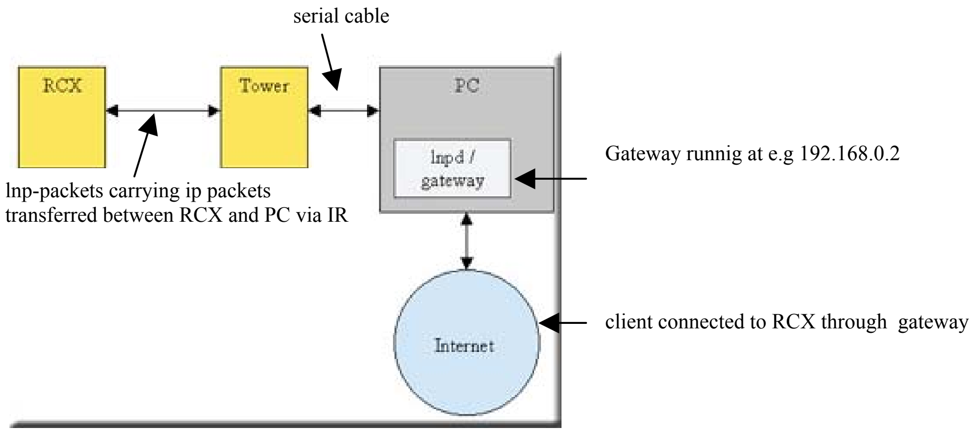
4 TCP/IP and LNP together

4.1 Integration into the LegOS-kernel.

The idea to use the, unused, integrity layer as a transport media offers full advantage of all the mentioned features of the LNP-protocol implementation including collision detection with a minimal overhead of only 3 bytes, F0, LEN and CHK.
Using real, true TCP/IP on the RCX and a LNP-TCP/IP gateway on the PC gives us the unique opportunity to use every programming language we want that supports sockets and TCP/IP to easily write code to interact with the RCX from all over the world, if we want to.
Also, to continue using LNP offers full backward compatibility at the price of precious RAM.

4.2 TCP/IP and the LNP-deamon.

The lnpd is used to interact with the user application, acting as a TCP/IP-LNP/gateway.
On the RCX the uIP takes care of the packet transmission using the LNP integrity packets.


serial cable
RCX
Tower
PC
lnpd / gateway
Gateway runnig at e.g 192.168.0.2
lnp-packets carrying ip packets transferred between RCX and PC via IR
Internet
client connected to RCX through gateway

This diagram shows how the RCX and the internet can be connected.

5 Alternative approaches.

Instead of using the underlying LNP, it might be better to completely get rid of it and replace it with a tiny implementation of the SLIP-protocol.
The major drawback of this solution woud be a rewrite of both the program handler on the RCX and the according code ,on the PC., the code we use to put our code on the RCX.
The big advantage would be, that, because SLIP is available on a large number of platforms, including MS Windows and even the C64, we would not be forced to use either LINUX or the lnpd, because there would be no LNP anymore.

6 Performance

You cannot expect great performance from a system like the RCX. The maximum speed is 4800 bps. I was really surprised, that packet losses are usually only between 0 and 5 percent if the range of 3 meters is not exeeded.
An example of sending a few pings to the rcx is shown below. These pings give an idea of what one can expect when transferring data between the PC and the RCX using a tcp/ip connection.
These statistics were gathered during a test using the experimental setup shown in chapter 7, example of use.

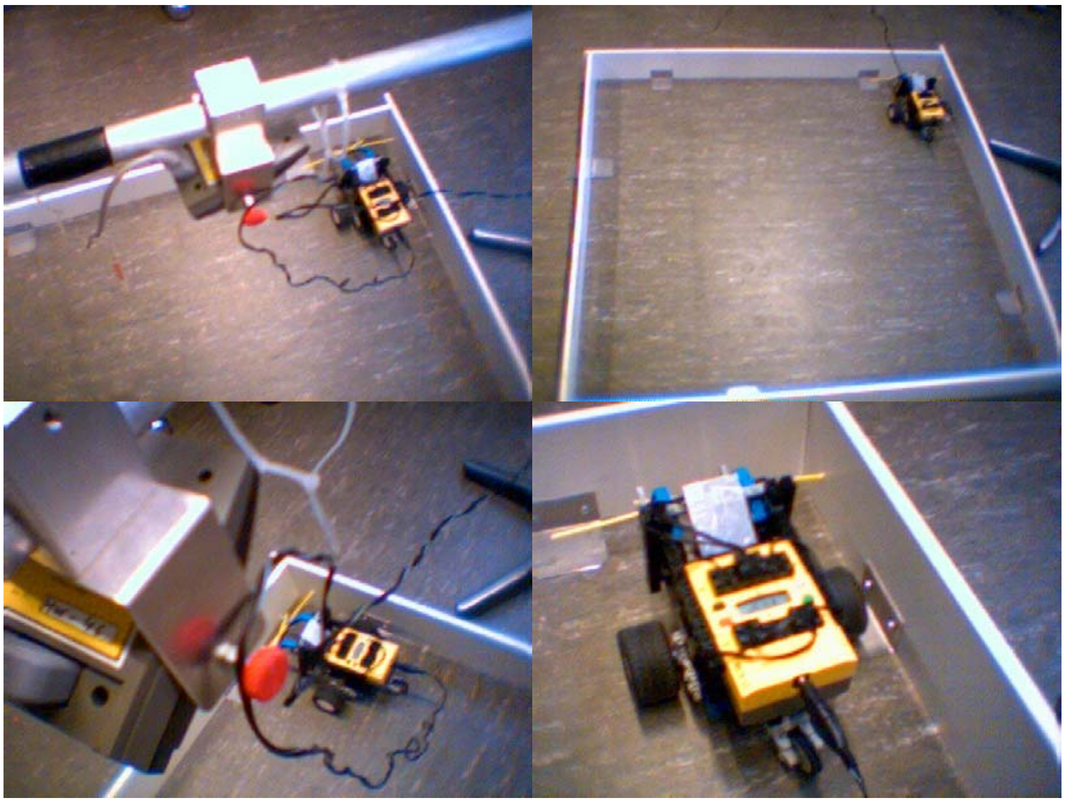
```
PING 192.168.0.2 (192.168.0.2): 56 data bytes
64 bytes from 192.168.0.2: icmp_seq=0 ttl=64 time=774.250 ms
64 bytes from 192.168.0.2: icmp_seq=1 ttl=64 time=782.528 ms
64 bytes from 192.168.0.2: icmp_seq=2 ttl=64 time=770.396 ms
64 bytes from 192.168.0.2: icmp_seq=3 ttl=64 time=778.150 ms
64 bytes from 192.168.0.2: icmp_seq=4 ttl=64 time=765.862 ms
64 bytes from 192.168.0.2: icmp_seq=5 ttl=64 time=773.645 ms
64 bytes from 192.168.0.2: icmp_seq=6 ttl=64 time=780.980 ms
64 bytes from 192.168.0.2: icmp_seq=7 ttl=64 time=768.727 ms
64 bytes from 192.168.0.2: icmp_seq=8 ttl=64 time=776.567 ms
64 bytes from 192.168.0.2: icmp_seq=9 ttl=64 time=765.117 ms
64 bytes from 192.168.0.2: icmp_seq=10 ttl=64 time=772.871 ms
64 bytes from 192.168.0.2: icmp_seq=11 ttl=64 time=780.651 ms
64 bytes from 192.168.0.2: icmp_seq=12 ttl=64 time=767.981 ms
64 bytes from 192.168.0.2: icmp_seq=13 ttl=64 time=775.778 ms
64 bytes from 192.168.0.2: icmp_seq=14 ttl=64 time=764.419 ms
64 bytes from 192.168.0.2: icmp_seq=15 ttl=64 time=772.142 ms
64 bytes from 192.168.0.2: icmp_seq=16 ttl=64 time=779.883 ms
64 bytes from 192.168.0.2: icmp_seq=17 ttl=64 time=768.426 ms
64 bytes from 192.168.0.2: icmp_seq=18 ttl=64 time=776.325 ms
64 bytes from 192.168.0.2: icmp_seq=19 ttl=64 time=764.031 ms
64 bytes from 192.168.0.2: icmp_seq=20 ttl=64 time=771.778 ms

--- 192.168.0.2 ping statistics ---
22 packets transmitted, 21 packets received, 4% packet loss
round-trip min/avg/max = 764.031/772.881/782.528 ms
```

In general TCP/IP mechanisms will also degrade the throughput, whereas the additional 3 bytes overhead can be neglected.
When transferring data between the RCX and the PC, things like maximum throughput should be something not to be concerned of. The only thing one should be concern about is reliability, and this is taken care of by the TCP/IP-protocol.
Using the uIP stack on the RCX is leaving us only with a couple of kbytes for own code., therefore transferring large amounts of data into the RCX's RAM is out of the question. For other purposes such transferring as sensor data or information to remote control the RCX, e.g. over 128 bytes of payload per packet at 2400 bps are still a huge amount of data, if no bit gets wasted.

7 Example of use

One of the main purposes of having a tcp/ip enabled RCX is to maintain and ensure a reliable connection between the PC and the robot.
The following pictures show a robot using a tcp/ip enabled RCX.
The robot has initially been developed by Philip Porschke as a part of his report. His thesis is about robots using genetic algorithms, helping the robot to quickly adapt to the given environment. To send the new code to the RCX a safe connection oriented protocol such as tcp/ip is inherently required. Tests have shown that uIP could be used to succesfully achieve this task as long as the RCX is moving within a one square meter rectangle as shown below.
This experimental setup helps keeping the packet losses at a negligible level (below 5%).



The experimental setup shown on the pictures is used to demonstrate that wherever the robot is located within the boundaries a reliable connetion can be maintained. Using ping to check for packet losses has proved, that occasional packet losses are between 0-5%. The transceiver is mounted above the square at about 1,30 meter.
Because, normally the RCX's transceiver transmits horizontally whereas the transceiver (aka. tower) is transmitting vertically, a small reflector is mounted to the RCX to fix that problem.

8 Summary

This student research project was about setting up a reliable connection between the PC and the RCX.
I chose using tcp/ip to achieve this task.
Using tcp/ip on the RCX and legOS was something almost everybody, including Martin Cornelius who has written parts of lnp, has considered to be an impossible task and tcp/ip on the RCX was considered to be way too much for the H8 microcontroller.

But, because there already were extremely small implementations of tcp/ip stacks on the web i was very optimistic that it could indeed be done.

Originally, after searching the web and eventually finding out about lwIP, a small stack also written by Adam Dunkels, i considered writing a smaller version of lwIP myself, which, after dicussing the project with Adam became absolete, because he was so kind sending me uIP, a smaller version of lwIP.uIp is specifically designed for my project.
I have to say that i have achieved everything that i wanted to and exactly how i wanted it to be.

This student research project has given me a very detailed insight into the problems regarding very small systems.especially when dealing with such cheap harware like the RCX.

I ve also learned a lot about the problems occuring in wireless networks and the low level details of TCP/IP and the performance of TCP/IP and UDP in wireless networks.
Surprisingly, the problems have mostly occurred on the Linux side. At first i thought the RCX would be the major problem in terms of RAM, but linking the stack to the kernel has shown that running a TCP/IP stack on such a small system is in fact.possible.
Using the stack, removing the sound support and downloading the rcx_uip.lx code is leaving us with about 3KB of free memory.
In a world where 512 MB of RAM is becoming or already has become a standard,one might say that 3 KB of RAM are not enough to write non-trivial code,but you should keep in mind that the whole stack itself is only about 3 KB. 3 KB is in fact still a lot of RAM if used wisely.

9 Future work

The future work definetely should be the integration of SLIP and the mentioned changes.
Writing code for the uIP without recompiling the whole kernel would also be very nice.
It would also be nice to see my implementation in the next legOS release.

Appendix

A TCP/IP

The following pages give a very brief, compact introduction and explanation of TCP/IP. There are a lot of books around explaining TCP/IP indepth, but it might be handy to have a short, yet easy to understand reference of TCP/IP and a few details beyond basic TCP/IP together with this report.

What is TCP/IP?

TCP/IP is a set of protocols developed to allow cooperating computers to share resources across a network. It was developed by a community of researchers centered around the ARPAnet. Certainly the ARPAnet is the best-known TCP/IP network. However as of June, 87, at least 130 different vendors had products that support TCP/IP, and thousands of networks of all kinds use it.

First some basic definitions. The most accurate name for the set of protocols we are describing is the "Internet protocol suite". TCP and IP are two of the protocols in this suite. (They will be described below.) Because TCP and IP are the best known of the protocols, it has become common to use the term TCP/IP or IP/TCP to refer to the whole family. It is probably not worth fighting this habit. However this can lead to some oddities. For example, I find myself talking about NFS as being based on TCP/IP, even though it doesn't use TCP at all. (It does use IP. But it uses an alternative protocol, UDP, instead of TCP. All of this alphabet soup will be unscrambled in the following pages.)

The Internet is a collection of networks, including the Arpanet, NSFnet, regional networks such as NYsernet, local networks at a number of University and research institutions, and a number of military networks. The term "Internet" applies to this entire set of networks. The subset of them that is managed by the Department of Defense is referred to as the "DDN" (Defense Data Network). This includes some research-oriented networks, such as the Arpanet, as well as more strictly military ones. (Because much of the funding for Internet protocol developments is done via the DDN organization, the terms Internet and DDN can sometimes seem equivalent.) All of these networks are connected to each other. Users can send messages from any of them to any other, except where there are security or other policy restrictions on access. Officially speaking, the Internet protocol documents are simply standards adopted by the Internet community for its own use. More recently, the Department of Defense issued a MILSPEC definition of TCP/IP. This was intended to be a more formal definition, appropriate for use in purchasing specifications. However most of the TCP/IP community continues to use the Internet standards. The MILSPEC version is intended to be consistent with it.

Whatever it is called, TCP/IP is a family of protocols. A few provide "low-level" functions needed for many applications. These include IP, TCP, and UDP. (These will be described in a bit more detail later.) Others are protocols for doing specific tasks, e.g. transferring files between computers, sending mail, or finding out who is logged in on another computer. Initially TCP/IP was used mostly between minicomputers or mainframes. These machines had their own disks, and generally were self-contained. Thus the most important "traditional" TCP/IP services are:

- file transfer. The file transfer protocol (FTP) allows a user on any computer to get files from another computer, or to send files to another computer. Security is handled by requiring the user to specify a user name and password for the other computer. Provisions are made for handling file transfer between machines with different character set, end of line conventions, etc. This is not quite the same thing as more recent "network file system" or "netbios" protocols, which will be described below. Rather, FTP is a utility that you run any time you want to access a file on another system. You use it to copy the file to your own system. You then work with the local copy. (See RFC 959 for specifications for FTP.)
- remote login. The network terminal protocol (TELNET) allows a user to log in on any other computer on the network. You start a remote session by specifying a computer to connect to. From that time until you finish the session, anything you type is sent to the other computer. Note that you are really still talking to your own computer. But the telnet program effectively makes your computer invisible while it is running. Every character you type is sent directly to the other system. Generally, the connection to the remote computer behaves much like a dialup connection. That is, the remote system will ask you to log in and give a password, in whatever manner it would normally ask a user who had just dialed it up. When you log off of the other computer, the telnet program exits, and you will find yourself talking to

your own computer. Microcomputer implementations of telnet generally include a terminal emulator for some common type of terminal. (See RFC's 854 and 855 for specifications for telnet. By the way, the telnet protocol should not be confused with Telenet, a vendor of commercial network services.)

- computer mail. This allows you to send messages to users on other computers. Originally, people tended to use only one or two specific computers. They would maintain "mail files" on those machines. The computer mail system is simply a way for you to add a message to another user's mail file. There are some problems with this in an environment where microcomputers are used. The most serious is that a micro is not well suited to receive computer mail. When you send mail, the mail software expects to be able to open a connection to the addressee's computer, in order to send the mail. If this is a microcomputer, it may be turned off, or it may be running an application other than the mail system. For this reason, mail is normally handled by a larger system, where it is practical to have a mail server running all the time. Microcomputer mail software then becomes a user interface that retrieves mail from the mail server. (See RFC 821 and 822 for specifications for computer mail. See RFC 937 for a protocol designed for microcomputers to use in reading mail from a mail server.)

These services should be present in any implementation of TCP/IP, except that micro-oriented implementations may not support computer mail. These traditional applications still play a very important role in TCP/IP-based networks. However more recently, the way in which networks are used has been changing. The older model of a number of large, self-sufficient computers is beginning to change. Now many installations have several kinds of computers, including microcomputers, workstations, minicomputers, and mainframes. These computers are likely to be configured to perform specialized tasks. Although people are still likely to work with one specific computer, that computer will call on other systems on the net for specialized services. This has led to the "server/client" model of network services. A server is a system that provides a specific service for the rest of the network. A client is another system that uses that service. (Note that the server and client need not be on different computers. They could be different programs running on the same computer.) Here are the kinds of servers typically present in a modern computer setup. Note that these computer services can all be provided within the framework of TCP/IP.

- network file systems. This allows a system to access files on another computer in a somewhat more closely integrated fashion than FTP. A network file system provides the illusion that disks or other devices from one system are directly connected to other systems. There is no need to use a special network utility to access a file on another system. Your computer simply thinks it has some extra disk drives. These extra "virtual" drives refer to the other system's disks. This capability is useful for several different purposes. It lets you put large disks on a few computers, but still give others access to the disk space. Aside from the obvious economic benefits, this allows people working on several computers to share common files. It makes system maintenance and backup easier, because you don't have to worry about updating and backing up copies on lots of different machines. A number of vendors now offer high-performance diskless computers. These computers have no disk drives at all. They are entirely dependent upon disks attached to common "file servers". (See RFC's 1001 and 1002 for a description of PC-oriented NetBIOS over TCP. In the workstation and minicomputer area, Sun's Network File System is more likely to be used. Protocol specifications for it are available from Sun Microsystems.)
- remote printing. This allows you to access printers on other computers as if they were directly attached to yours. (The most commonly used protocol is the remote lineprinter protocol from Berkeley Unix. Unfortunately, there is no protocol document for this. However the C code is easily obtained from Berkeley, so implementations are common.)
- remote execution. This allows you to request that a particular program be run on a different computer. This is useful when you can do most of your work on a small computer, but a few tasks require the resources of a larger system. There are a number of different kinds of remote execution. Some operate on a command by command basis. That is, you request that a specific command or set of commands should run on some specific computer. (More sophisticated versions will choose a system that happens to be free.) However there are also "remote procedure call" systems that allow a program to call a subroutine that will run on another computer. (There are many protocols of this sort. Berkeley Unix contains two servers to execute commands remotely: rsh and rexec. The man pages describe the protocols that they use. The user-contributed software with Berkeley 4.3 contains a "distributed shell" that will distribute tasks among a set of systems, depending upon load. Remote procedure call mechanisms have been a topic for research for a number of years, so many organizations have implementations of such facilities. The most widespread commercially-supported remote procedure call protocols seem to be Xerox's Courier and Sun's RPC. Protocol documents are available from Xerox and Sun. There is a public implementation of Courier over TCP as part of the user-contributed software with Berkeley 4.3. An implementation of RPC was posted to Usenet by Sun, and also appears as part of the user-contributed software with Berkeley 4.3.)

- name servers. In large installations, there are a number of different collections of names that have to be managed. This includes users and their passwords, names and network addresses for computers, and accounts. It becomes very tedious to keep this data up to date on all of the computers. Thus the databases are kept on a small number of systems. Other systems access the data over the network. (RFC 822 and 823 describe the name server protocol used to keep track of host names and Internet addresses on the Internet. This is now a required part of any TCP/IP implementation. IEN 116 describes an older name server protocol that is used by a few terminal servers and other products to look up host names. Sun's Yellow Pages system is designed as a general mechanism to handle user names, file sharing groups, and other databases commonly used by Unix systems. It is widely available commercially. Its protocol definition is available from Sun.)
- terminal servers. Many installations no longer connect terminals directly to computers. Instead they connect them to terminal servers. A terminal server is simply a small computer that only knows how to run telnet (or some other protocol to do remote login). If your terminal is connected to one of these, you simply type the name of a computer, and you are connected to it. Generally it is possible to have active connections to more than one computer at the same time. The terminal server will have provisions to switch between connections rapidly, and to notify you when output is waiting for another connection. (Terminal servers use the telnet protocol, already mentioned. However any real terminal server will also have to support name service and a number of other protocols.)
- network-oriented window systems. Until recently, high- performance graphics programs had to execute on a computer that had a bit-mapped graphics screen directly attached to it. Network window systems allow a program to use a display on a different computer. Full-scale network window systems provide an interface that lets you distribute jobs to the systems that are best suited to handle them, but still give you a single graphically-based user interface. (The most widely-implemented window system is X. A protocol description is available from MIT's Project Athena. A reference implementation is publically available from MIT. A number of vendors are also supporting NeWS, a window system defined by Sun. Both of these systems are designed to use TCP/IP.)

Note that some of the protocols described above were designed by Berkeley, Sun, or other organizations. Thus they are not officially part of the Internet protocol suite. However they are implemented using TCP/IP, just as normal TCP/IP application protocols are. Since the protocol definitions are not considered proprietary, and since commercially-support implementations are widely available, it is reasonable to think of these protocols as being effectively part of the Internet suite. Note that the list above is simply a sample of the sort of services available through TCP/IP. However it does contain the majority of the "major" applications. The other commonly-used protocols tend to be specialized facilities for getting information of various kinds, such as who is logged in, the time of day, etc. However if you need a facility that is not listed here, we encourage you to look through the current edition of Internet Protocols (currently RFC 1011), which lists all of the available protocols, and also to look at some of the major TCP/IP implementations to see what various vendors have added.

General description of the TCP/IP protocols

TCP/IP is a layered set of protocols. In order to understand what this means, it is useful to look at an example. A typical situation is sending mail. First, there is a protocol for mail. This defines a set of commands which one machine sends to another, e.g. commands to specify who the sender of the message is, who it is being sent to, and then the text of the message. However this protocol assumes that there is a way to communicate reliably between the two computers. Mail, like other application protocols, simply defines a set of commands and messages to be sent. It is designed to be used together with TCP and IP. TCP is responsible for making sure that the commands get through to the other end. It keeps track of what is sent, and retransmitts anything that did not get through. If any message is too large for one datagram, e.g. the text of the mail, TCP will split it up into several datagrams, and make sure that they all arrive correctly. Since these functions are needed for many applications, they are put together into a separate protocol, rather than being part of the specifications for sending mail. You can think of TCP as forming a library of routines that applications can use when they need reliable network communications with another computer. Similarly, TCP calls on the services of IP. Although the services that TCP supplies are needed by many applications, there are still some kinds of applications that don't need them. However there are some services that every application needs. So these services are put together into IP. As with TCP, you can think of IP as a library of routines that TCP calls on, but which is also available to applications that don't use TCP. This strategy of building several levels of protocol is called "layering". We think of the applications programs such as mail, TCP, and IP, as being separate "layers", each of which calls on the services of the layer below it. Generally, TCP/IP applications use 4 layers:

- an application protocol such as mail
- a protocol such as TCP that provides services need by many applications
- IP, which provides the basic service of getting datagrams to their destination
- the protocols needed to manage a specific physical medium, such as Ethernet or a point to point line.

TCP/IP is based on the "catenet model". (This is described in more detail in IEN 48.) This model assumes that there are a large number of independent networks connected together by gateways. The user should be able to access computers or other resources on any of these networks. Datagrams will often pass through a dozen different networks before getting to their final destination. The routing needed to accomplish this should be completely invisible to the user. As far as the user is concerned, all he needs to know in order to access another system is an "Internet address". This is an address that looks like 128.6.4.194. It is actually a 32-bit number. However it is normally written as 4 decimal numbers, each representing 8 bits of the address. (The term "octet" is used by Internet documentation for such 8-bit chunks. The term "byte" is not used, because TCP/IP is supported by some computers that have byte sizes other than 8 bits.) Generally the structure of the address gives you some information about how to get to the system. For example, 128.6 is a network number assigned by a central authority to Rutgers University. Rutgers uses the next octet to indicate which of the campus Ethernets is involved. 128.6.4 happens to be an Ethernet used by the Computer Science Department. The last octet allows for up to 254 systems on each Ethernet. (It is 254 because 0 and 255 are not allowed, for reasons that will be discussed later.) Note that 128.6.4.194 and 128.6.5.194 would be different systems. The structure of an Internet address is described in a bit more detail later.

Of course we normally refer to systems by name, rather than by Internet address. When we specify a name, the network software looks it up in a database, and comes up with the corresponding Internet address. Most of the network software deals strictly in terms of the address. (RFC 882 describes the name server technology used to handle this lookup.)

TCP/IP is built on "connectionless" technology. Information is transfered as a sequence of "datagrams". A datagram is a collection of data that is sent as a single message. Each of these datagrams is sent through the network individually. There are provisions to open connections (i.e. to start a conversation that will continue for some time). However at some level, information from those connections is broken up into datagrams, and those datagrams are treated by the network as completely separate. For example, suppose you want to transfer a 15000 octet file. Most networks can't handle a 15000 octet datagram. So the protocols will break this up into something like 30 500-octet datagrams. Each of these datagrams will be sent to the other end. At that point, they will be put back together into the 15000-octet file. However while those datagrams are in transit, the network doesn't know that there is any connection between them. It is perfectly possible that datagram 14 will actually arrive before datagram 13. It is also possible that somewhere in the network, an error will occur, and some datagram won't get through at all. In that case, that datagram has to be sent again.

Note by the way that the terms "datagram" and "packet" often seem to be nearly interchangable. Technically, datagram is the right word to use when describing TCP/IP. A datagram is a unit of data, which is what the protocols deal with. A packet is a physical thing, appearing on an Ethernet or some wire. In most cases a packet simply contains a datagram, so there is very little difference. However they can differ. When TCP/IP is used on top of X.25, the X.25 interface breaks the datagrams up into 128-byte packets. This is invisible to IP, because the packets are put back together into a single datagram at the other end before being processed by TCP/IP. So in this case, one IP datagram would be carried by several packets. However with most media, there are efficiency advantages to sending one datagram per packet, and so the distinction tends to vanish.

The TCP level

Two separate protocols are involved in handling TCP/IP datagrams. TCP (the "transmission control protocol") is responsible for breaking up the message into datagrams, reassembling them at the other end, resending anything that gets lost, and putting things back in the right order. IP (the "internet protocol") is responsible for routing individual datagrams. It may seem like TCP is doing all the work. And in small networks that is true. However in the Internet, simply getting a datagram to its destination can be a complex job. A connection may require the datagram to go through several networks at Rutgers, a serial line to the John von Neuman Supercomputer Center, a couple of Ethernets there, a series of 56Kbaud phone lines to another NSFnet site, and more Ethernets on another campus. Keeping track of the routes to all of the destinations and handling incompatibilities among different transport media turns out to be a complex job. Note that the interface between TCP and IP is fairly simple. TCP simply hands IP a datagram with a destination. IP doesn't know how this datagram relates to any datagram before it or after it.

It may have occurred to you that something is missing here. We have talked about Internet addresses, but not about how you keep track of multiple connections to a given system. Clearly it isn't enough to get a datagram to the right destination. TCP has to know which connection this datagram is part of. This task is referred to as "demultiplexing." In fact, there are several levels of demultiplexing going on in TCP/IP. The information needed to do this demultiplexing is contained in a series of "headers". A header is simply a few extra octets tacked onto the beginning of a datagram by some protocol in order to keep track of it. It's a lot like putting a letter into an envelope and putting an address on the outside of the envelope. Except with modern networks it happens several times. It's like you put the letter into a little envelope, your secretary puts that into a somewhat bigger envelope, the campus mail center puts that envelope into a still bigger one, etc. Here is an overview of the headers that get stuck on a message that passes through a typical TCP/IP network:

We start with a single data stream, say a file you are trying to send to some other computer:

```
......................................................
```

TCP breaks it up into manageable chunks. (In order to do this, TCP has to know how large a datagram your network can handle. Actually, the TCP's at each end say how big a datagram they can handle, and then they pick the smallest size.)

```
....   ....   ....   ....   ....   ....   ....   ....
```

TCP puts a header at the front of each datagram. This header actually contains at least 20 octets, but the most important ones are a source and destination "port number" and a "sequence number". The port numbers are used to keep track of different conversations. Suppose 3 different people are transferring files. Your TCP might allocate port numbers 1000, 1001, and 1002 to these transfers. When you are sending a datagram, this becomes the "source" port number, since you are the source of the datagram. Of course the TCP at the other end has assigned a port number of its own for the conversation. Your TCP has to know the port number used by the other end as well. (It finds out when the connection starts, as we will explain below.) It puts this in the "destination" port field. Of course if the other end sends a datagram back to you, the source and destination port numbers will be reversed, since then it will be the source and you will be the destination. Each datagram has a sequence number. This is used so that the other end can make sure that it gets the datagrams in the right order, and that it hasn't missed any. (See the TCP specification for details.) TCP doesn't number the datagrams, but the octets. So if there are 500 octets of data in each datagram, the first datagram might be numbered 0, the second 500, the next 1000, the next 1500, etc. Finally, I will mention the Checksum. This is a number that is computed by adding up all the octets in the datagram (more or less - see the TCP spec). The result is put in the header. TCP at the other end computes the checksum again. If they disagree, then something bad happened to the datagram in transmission, and it is thrown away. So here's what the datagram looks like now.

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   your data ... next 500 octets                               |
|   ......                                                      |
```

If we abbreviate the TCP header as "T", the whole file now looks like this:

```
T....   T....   T....   T....   T....   T....   T....
```

You will note that there are items in the header that I have not described above. They are generally involved with managing the connection. In order to make sure the datagram has arrived at its destination, the recipient has to send back an "acknowledgement". This is a datagram whose "Acknowledgement number" field is filled in. For example, sending a packet with an acknowledgement of 1500 indicates that you have received all the data up to octet number 1500. If the sender doesn't get an acknowledgement within a reasonable amount of time, it sends the data again. The window is used to control how much data can be in transit at any one time. It is not practical to wait for each datagram to be acknowledged before sending the next one. That would slow things down too much. On the other hand, you can't just keep sending, or a fast computer might overrun the capacity of a slow one to absorb data. Thus each end indicates how much new data it is currently prepared to absorb by putting the number of octets in its "Window" field. As the computer receives data, the amount of space left in its window decreases. When it goes to zero, the sender has to stop. As the receiver processes the data, it increases its window, indicating that it is ready to accept more data. Often the same datagram can be used to acknowledge receipt of a set of data and to give permission for additional new data (by an updated window). The "Urgent" field allows one end to tell the other to skip ahead in its processing to a particular octet. This is often useful for handling asynchronous events, for example when you type a control character or other command that interrupts output. The other fields are beyond the scope of this document.

The IP level

TCP sends each of these datagrams to IP. Of course it has to tell IP the Internet address of the computer at the other end. Note that this is all IP is concerned about. It doesn't care about what is in the datagram, or even in the TCP header. IP's job is simply to find a route for the datagram and get it to the other end. In order to allow gateways or other intermediate systems to forward the datagram, it adds its own header. The main things in this header are the source and destination Internet address (32-bit addresses, like 128.6.4.194), the protocol number, and another checksum. The source Internet address is simply the address of your machine. (This is necessary so the other end knows where the datagram came from.) The destination Internet address is the address of the other machine. (This is necessary so any gateways in the middle know where you want the datagram to go.) The protocol number tells IP at the other end to send the datagram to TCP. Although most IP traffic uses TCP, there are other protocols that can use IP, so you have to tell IP which protocol to send the datagram to. Finally, the checksum allows IP at the other end to verify that the header wasn't damaged in transit. Note that TCP and IP have separate checksums. IP needs to be able to verify that the header didn't get damaged in transit, or it could send a message to the wrong place. For reasons not worth discussing here, it is both more efficient and safer to have TCP compute a separate checksum for the TCP header and data. Once IP has tacked on its header, here's what the message looks like:

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  TCP header, then your data ......                            |
|                                                               |
```

If we represent the IP header by an "I", your file now looks like this:

```
   IT....   IT....   IT....   IT....   IT....   IT....   IT....
```

Again, the header contains some additional fields that have not been discussed. Most of them are beyond the scope of this document. The flags and fragment offset are used to keep track of the pieces when a datagram has to be split up. This can happen when datagrams are forwarded through a network for which they are too big. (This will be discussed a bit more below.) The time to live is a number that is decremented whenever the datagram passes through a system. When it goes to zero, the datagram is discarded. This is done in case a loop develops in the system somehow. Of course this should be impossible, but well-designed networks are built to cope with "impossible" conditions.

At this point, it's possible that no more headers are needed. If your computer happens to have a direct phone line connecting it to the destination computer, or to a gateway, it may simply send the datagrams out on the line (though likely a synchronous protocol such as HDLC would be used, and it would add at least a few octets at the beginning and end).

The Ethernet level

However most of our networks these days use Ethernet. So now we have to describe Ethernet's headers. Unfortunately, Ethernet has its own addresses. The people who designed Ethernet wanted to make sure that no two machines would end up with the same Ethernet address. Furthermore, they didn't want the user to have to worry about assigning addresses. So each Ethernet controller comes with an address builtin from the factory. In order to make sure that they would never have to reuse addresses, the Ethernet designers allocated 48 bits for the Ethernet address. People who make Ethernet equipment have to register with a central authority, to make sure that the numbers they assign don't overlap any other manufacturer. Ethernet is a "broadcast medium". That is, it is in effect like an old party line telephone. When you send a packet out on the Ethernet, every machine on the network sees the packet. So something is needed to make sure that the right machine gets it. As you might guess, this involves the Ethernet header. Every Ethernet packet has a 14-octet header that includes the source and destination Ethernet address, and a type code. Each machine is supposed to pay attention only to packets with its own Ethernet address in the destination field. (It's perfectly possible to cheat, which is one reason that Ethernet communications are not terribly secure.) Note that there is no connection between the Ethernet address and the Internet address. Each machine has to have a table of what Ethernet address corresponds to what Internet address. (We will describe how this table is constructed a bit later.) In addition to the addresses, the header contains a type code. The type code is to allow for several different protocol families to be used on the same network. So you can use TCP/IP, DECnet, Xerox NS, etc. at the same time. Each of them will put a different value in the type field. Finally, there is a checksum. The Ethernet controller computes a checksum of the entire packet. When the other end receives the packet, it recomputes the checksum, and throws the packet away if the answer disagrees with the original. The checksum is put on the end of the packet, not in the header. The final result is that your message looks like this:

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       Ethernet destination address (first 32 bits)            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Ethernet dest (last 16 bits)  |Ethernet source (first 16 bits)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       Ethernet source address (last 32 bits)                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Type code              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  IP header, then TCP header, then your data                   |
|                                                               |
    ...
|                                                               |
|   end of your data                                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Ethernet Checksum                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

If we represent the Ethernet header with "E", and the Ethernet checksum with "C", your file now looks like this:

```
  EIT....C   EIT....C   EIT....C   EIT....C   EIT....C
```

When these packets are received by the other end, of course all the headers are removed. The Ethernet interface removes the Ethernet header and the checksum. It looks at the type code. Since the type code is the one assigned to IP, the Ethernet device driver passes the datagram up to IP. IP removes the IP header. It looks at the IP protocol field. Since the protocol type is TCP, it passes the datagram up to TCP. TCP now looks at the sequence number. It uses the sequence numbers and other information to combine all the datagrams into the original file.

The ends our initial summary of TCP/IP. There are still some crucial concepts we haven't gotten to, so we'll now go back and add details in several areas. (For detailed descriptions of the items discussed here see, RFC 793 for TCP, RFC 791 for IP, and RFC's 894 and 826 for sending IP over Ethernet.)

So far, we have described how a stream of data is broken up into datagrams, sent to another computer, and put back together. However something more is needed in order to accomplish anything useful. There has to be a way for you to open a connection to a specified computer, log into it, tell it what file you want, and control the transmission of the file. (If you have a different application in mind, e.g. computer mail, some analogous protocol is needed.) This is done by "application protocols". The application protocols run "on top" of TCP/IP. That is, when they want to send a message, they give the message to TCP. TCP makes sure it gets delivered to the other end. Because TCP and IP take care of all the networking details, the applications protocols can treat a network connection as if it were a simple byte stream, like a terminal or phone line.

Before going into more details about applications programs, we have to describe how you find an application. Suppose you want to send a file to a computer whose Internet address is 128.6.4.7. To start the process, you need more than just the Internet address. You have to connect to the FTP server at the other end. In general, network programs are specialized for a specific set of tasks. Most systems have separate programs to handle file transfers, remote terminal logins, mail, etc. When you connect to 128.6.4.7, you have to specify that you want to talk to the FTP server. This is done by having "well-known sockets" for each server. Recall that TCP uses port numbers to keep track of individual conversations. User programs normally use more or less random port numbers. However specific port numbers are assigned to the programs that sit waiting for requests. For example, if you want to send a file, you will start a program called "ftp". It will open a connection using some random number, say 1234, for the port number on its end. However it will specify port number 21 for the other end. This is the official port number for the FTP server. Note that there are two different programs involved. You run ftp on your side. This is a program designed to accept commands from your terminal and pass them on to the other end. The program that you talk to on the other machine is the FTP server. It is designed to accept commands from the network connection, rather than an interactive terminal. There is no need for your program to use a well-known socket number for itself. Nobody is trying to find it. However the servers have to have well-known numbers, so that people can open connections to them and start sending them commands. The official port numbers for each program are given in "Assigned Numbers".

Note that a connection is actually described by a set of 4 numbers: the Internet address at each end, and the TCP port number at each end. Every datagram has all four of those numbers in it. (The Internet addresses are in the IP header, and the TCP port numbers are in the TCP header.) In order to keep things straight, no two connections can have the same set of numbers. However it is enough for any one number to be different. For example, it is perfectly possible for two different users on a machine to be sending files to the same other machine. This could result in connections with the following parameters:

	Internet addresses	TCP ports
connection 1	128.6.4.194, 128.6.4.7	1234, 21
connection 2	128.6.4.194, 128.6.4.7	1235, 21

Since the same machines are involved, the Internet addresses are the same. Since they are both doing file transfers, one end of the connection involves the well-known port number for FTP. The only thing that differs is the port number for the program that the users are running. That's enough of a difference. Generally, at least one end of the connection asks the network software to assign it a port number that is guaranteed to be unique. Normally, it's the user's end, since the server has to use a well-known number.

Now that we know how to open connections, let's get back to the applications programs. As mentioned earlier, once TCP has opened a connection, we have something that might as well be a simple wire. All the hard parts are handled by TCP and IP. However we still need some agreement as to what we send over this connection. In effect this is simply an agreement on what set of commands the application will understand, and the format in which they are to be sent. Generally, what is sent is a combination of commands and data. They use context to differentiate. For example, the mail protocol works like this: Your mail program opens a connection to the mail server at the other end. Your program gives it your machine's name, the sender of the message, and the recipients you want it sent to. It then sends a command saying that it is starting the message. At that point, the other end stops treating what it sees as commands, and starts accepting the message. Your end then starts sending the text of the message. At the end of the message, a special mark is sent (a dot in the first column). After that, both ends understand that your program is again sending commands. This is the simplest way to do things, and the one that most applications use.

File transfer is somewhat more complex. The file transfer protocol involves two different connections. It starts out just like mail. The user's program sends commands like "log me in as this user", "here is my password", "send me the file with this name". However once the command to send data is sent, a second connection is opened for the data itself. It would certainly be possible to send the data on the same connection, as mail does. However file transfers often take a long time. The designers of the file transfer protocol wanted to allow the user to continue issuing commands while the transfer is going on. For example, the user might make an inquiry, or he might abort the transfer. Thus the designers felt it was best to use a separate connection for the data and leave the original command connection for commands. (It is also possible to open command connections to two different computers, and tell them to send a file from one to the other. In that case, the data couldn't go over the command connection.)

Remote terminal connections use another mechanism still. For remote logins, there is just one connection. It normally sends data. When it is necessary to send a command (e.g. to set the terminal type or to change some mode), a special character is used to indicate that the next character is a command. If the user happens to type that special character as data, two of them are sent.

We are not going to describe the application protocols in detail in this document. It's better to read the RFC's yourself. However there are a couple of common conventions used by applications that will be described here. First, the common network representation: TCP/IP is intended to be usable on any computer. Unfortunately, not all computers agree on how data is represented. There are differences in character codes (ASCII vs. EBCDIC), in end of line conventions (carriage return, line feed, or a representation using counts), and in whether terminals expect characters to be sent individually or a line at a time. In order to allow computers of different kinds to communicate, each applications protocol defines a standard representation. Note that TCP and IP do not care about the representation. TCP simply sends octets. However the programs at both ends have to agree on how the octets are to be interpreted. The RFC for each application specifies the standard representation for that application. Normally it is "net ASCII". This uses ASCII characters, with end of line denoted by a carriage return followed by a line feed. For remote login, there is also a definition of a "standard terminal", which turns out to be a half-duplex terminal with echoing happening on the local machine. Most applications also make provisions for the two computers to agree on other representations that they may find more convenient. For example, PDP-10's have 36-bit words. There is a way that two PDP-10's can agree to send a 36-bit binary file. Similarly, two systems that prefer full-duplex terminal conversations can agree on that. However each application has a standard representation, which every machine must support.

An example application: SMTP

In order to give a bit better idea what is involved in the application protocols, I'm going to show an example of SMTP, which is the mail protocol. (SMTP is "simple mail transfer protocol.) We assume that a computer called TOPAZ.RUTGERS.EDU wants to send the following message.

```
Date: Sat, 27 Jun 87 13:26:31 EDT
From: hedrick@topaz.rutgers.edu
To: levy@red.rutgers.edu
Subject: meeting

Let's get together Monday at 1pm.
```

First, note that the format of the message itself is described by an Internet standard (RFC 822). The standard specifies the fact that the message must be transmitted as net ASCII (i.e. it must be ASCII, with carriage return/linefeed to delimit lines). It also describes the general structure, as a group of header lines, then a blank line, and then the body of the message. Finally, it describes the syntax of the header lines in detail. Generally they consist of a keyword and then a value.

Note that the addressee is indicated as LEVY@RED.RUTGERS.EDU. Initially, addresses were simply "person at machine". However recent standards have made things more flexible. There are now provisions for systems to handle other systems' mail. This can allow automatic forwarding on behalf of computers not connected to the Internet. It can be used to direct mail for a number of systems to one central mail server. Indeed there is no requirement that an actual computer by the name of RED.RUTGERS.EDU even exist. The name servers could be set up so that you mail to department names, and each department's mail is routed automatically to an appropriate computer. It is also possible that the part before the @ is something other than a user name. It is possible for programs to be set up to process mail. There are also provisions to handle mailing lists, and generic names such as "postmaster" or "operator".

The way the message is to be sent to another system is described by RFC's 821 and 974. The program that is going to be doing the sending asks the name server several queries to determine where to route the message. The first query is to find out which machines handle mail for the name RED.RUTGERS.EDU. In this case, the server replies that RED.RUTGERS.EDU handles its own mail. The program then asks for the address of

RED.RUTGERS.EDU, which is 128.6.4.2. Then the mail program opens a TCP connection to port 25 on 128.6.4.2. Port 25 is the well-known socket used for receiving mail. Once this connection is established, the mail program starts sending commands. Here is a typical conversation. Each line is labelled as to whether it is from TOPAZ or RED. Note that TOPAZ initiated the connection:

```
RED    220 RED.RUTGERS.EDU SMTP Service at 29 Jun 87 05:17:18 EDT
TOPAZ  HELO topaz.rutgers.edu
RED    250 RED.RUTGERS.EDU - Hello, TOPAZ.RUTGERS.EDU
TOPAZ  MAIL From:
RED    250 MAIL accepted
TOPAZ  RCPT To:
RED    250 Recipient accepted
TOPAZ  DATA
RED    354 Start mail input; end with .
TOPAZ  Date: Sat, 27 Jun 87 13:26:31 EDT
TOPAZ  From: hedrick@topaz.rutgers.edu
TOPAZ  To: levy@red.rutgers.edu
TOPAZ  Subject: meeting
TOPAZ
TOPAZ  Let's get together Monday at 1pm.
TOPAZ  .
RED    250 OK
TOPAZ  QUIT
RED    221 RED.RUTGERS.EDU
```

Service closing transmission channel First, note that commands all use normal text. This is typical of theInternet standards. Many of the protocols use standard ASCIIcommands. This makes it easy to watch what is going on and todiagnose problems. For example, the mail program keeps a log of eachconversation. If something goes wrong, the log file can simply be
mailed to the postmaster. Since it is normal text, he can see whatwas going on. It also allows a human to interact directly with the mail server, for testing. (Some newer protocols are complex enough that this is not practical. The commands would have to have a syntax that would require a significant parser. Thus there is a tendency for newer protocols to use binary formats. Generally they are structured like C or Pascal record structures.) Second, note that the responses all begin with numbers. This is also typical of Internet protocols. The allowable responses are defined in the protocol. The numbers allow the user program to respond unambiguously. The rest of the response is text, which is normally for use by any human who may be watching or looking at a log. It has no effect on the operation of the programs. (However there is one point at which the protocol uses part of the text of the response.) The commands themselves simply allow the mail program on one end to tell the mail server the information it needs to know in order to deliver the message. In this case, the mail server could get the information by looking at the message itself. But for more complex cases, that would not be safe. Every session must begin with a HELO, which gives the name of the system that initiated the connection. Then the sender and recipients are specified. (There can be more than one RCPT command, if there are several recipients.) Finally the data itself is sent. Note that the text of the message is terminated by a line containing just a period. (If such a line appears in the message, the period is doubled.) After the message is accepted, the sender can send another message, or terminate the session as in the example above. Generally, there is a pattern to the response numbers. The protocol defines the specific set of responses that can be sent as answers to any given command. However programs that don't want to analyze them in detail can just look at the first digit. In general, responses that begin with a 2 indicate success. Those that begin with 3 indicate that some further action is needed, as shown above. 4 and 5 indicate errors. 4 is a "temporary" error, such as a disk filling. The message should be saved, and tried again later. 5 is a permanent error, such as a non-existent recipient. The message should be returned to the sender with an error message. (For more details about the protocols mentioned in this section, see RFC's 821/822 for mail, RFC 959 for file transfer, and RFC's 854/855 for remote logins. For the well-known port numbers, see the current edition of Assigned Numbers, and possibly RFC 814.)

Protocols other than TCP: UDP and ICMP

So far, we have described only connections that use TCP. Recall that TCP is responsible for breaking up messages into datagrams, and reassembling them properly. However in many applications, we have messages that will always fit in a single datagram. An example is name lookup. When a user attempts to make a connection to another system, he will generally specify the system by name, rather than Internet address. His system has to translate that name to an address before it can do anything. Generally, only a few systems have the database used to translate names to addresses. So the user's system will want to send a query to one of the systems that has the database. This query is going to be very short. It will certainly fit in one datagram. So will the answer. Thus it seems silly to use TCP. Of course TCP does more than just break things up into datagrams. It also makes sure that the data arrives, resending datagrams where necessary. But for a question that fits in a single datagram, we don't need all the complexity of TCP to do this. If we don't get an answer after a few seconds, we can just ask again. For applications like this, there are alternatives to TCP.

The most common alternative is UDP ("user datagram protocol"). UDP is designed for applications where you don't need to put sequences of datagrams together. It fits into the system much like TCP. There is a UDP header. The network software puts the UDP header on the front of your data, just as it would put a TCP header on the front of your data. Then UDP sends the data to IP, which adds the IP header, putting UDP's protocol number in the protocol field instead of TCP's protocol number. However UDP doesn't do as much as TCP does. It doesn't split data into multiple datagrams. It doesn't keep track of what it has sent so it can resend if necessary. About all that UDP provides is port numbers, so that several programs can use UDP at once. UDP port numbers are used just like TCP port numbers. There are well-known port numbers for servers that use UDP. Note that the UDP header is shorter than a TCP header. It still has source and destination port numbers, and a checksum, but that's about it. No sequence number, since it is not needed. UDP is used by the protocols that handle name lookups (see IEN 116, RFC 882, and RFC 883), and a number of similar protocols.

Another alternative protocol is ICMP ("Internet control message protocol"). ICMP is used for error messages, and other messages intended for the TCP/IP software itself, rather than any particular user program. For example, if you attempt to connect to a host, your system may get back an ICMP message saying "host unreachable". ICMP can also be used to find out some information about the network. See RFC 792 for details of ICMP. ICMP is similar to UDP, in that it handles messages that fit in one datagram. However it is even simpler than UDP. It doesn't even have port numbers in its header. Since all ICMP messages are interpreted by the network software itself, no port numbers are needed to say where a ICMP message is supposed to go.

Keeping track of names and information: the domain system

As we indicated earlier, the network software generally needs a 32-bit Internet address in order to open a connection or send a datagram. However users prefer to deal with computer names rather than numbers. Thus there is a database that allows the software to look up a name and find the corresponding number. When the Internet was small, this was easy. Each system would have a file that listed all of the other systems, giving both their name and number. There are now too many computers for this approach to be practical. Thus these files have been replaced by a set of name servers that keep track of host names and the corresponding Internet addresses. (In fact these servers are somewhat more general than that. This is just one kind of information stored in the domain system.) Note that a set of interlocking servers are used, rather than a single central one. There are now so many different institutions connected to the Internet that it would be impractical for them to notify a central authority whenever they installed or moved a computer. Thus naming authority is delegated to individual institutions. The name servers form a tree, corresponding to institutional structure. The names themselves follow a similar structure. A typical example is the name BORAX.LCS.MIT.EDU. This is a computer at the Laboratory for Computer Science (LCS) at MIT. In order to find its Internet address, you might potentially have to consult 4 different servers. First, you would ask a central server (called the root) where the EDU server is. EDU is a server that keeps track of educational institutions. The root server would give you the names and Internet addresses of several servers for EDU. (There are several servers at each level, to allow for the possibly that one might be down.) You would then ask EDU where the server for MIT is. Again, it would give you names and Internet addresses of several servers for MIT. Generally, not all of those servers would be at MIT, to allow for the possibility of a general power failure at MIT. Then you would ask MIT where the server for LCS is, and finally you would ask one of the LCS servers about BORAX. The final result would be the Internet address for BORAX.LCS.MIT.EDU. Each of these levels is referred to as a "domain". The entire name, BORAX.LCS.MIT.EDU, is called a "domain name". (So are the names of the higher-level domains, such as LCS.MIT.EDU, MIT.EDU, and EDU.)

Fortunately, you don't really have to go through all of this most of the time. First of all, the root name servers also happen to be the name servers for the top-level domains such as EDU. Thus a single query to a root server will get you to MIT. Second, software generally remembers answers that it got before. So once we look up a name at LCS.MIT.EDU, our software remembers where to find servers for LCS.MIT.EDU, MIT.EDU, and EDU. It also remembers the translation of BORAX.LCS.MIT.EDU. Each of these pieces of information has a "time to live" associated with it. Typically this is a few days. After that, the information expires and has to be looked up again. This allows institutions to change things.

The domain system is not limited to finding out Internet addresses. Each domain name is a node in a database. The node can have records that define a number of different properties. Examples are Internet address, computer type, and a list of services provided by a computer. A program can ask for a specific piece of information, or all information about a given name. It is possible for a node in the database to be marked as an "alias" (or nickname) for another node. It is also possible to use the domain system to store information about users, mailing lists, or other objects.

There is an Internet standard defining the operation of these databases, as well as the protocols used to make queries of them. Every network utility has to be able to make such queries, since this is now the official way to evaluate host names. Generally utilities will talk to a server on their own system. This server will take care of contacting the other servers for them. This keeps down the amount of code that has to be in each application program.

The domain system is particularly important for handling computer mail. There are entry types to define what computer handles mail for a given name, to specify where an individual is to receive mail, and to define mailing lists.

(See RFC's 882, 883, and 973 for specifications of the domain system. RFC 974 defines the use of the domain system in sending mail.)

Routing

The description above indicated that the IP implementation is responsible for getting datagrams to the destination indicated by the destination address, but little was said about how this would be done. The task of finding how to get a datagram to its destination is referred to as "routing". In fact many of the details depend upon the particular implementation. However some general things can be said.

First, it is necessary to understand the model on which IP is based. IP assumes that a system is attached to some local network. We assume that the system can send datagrams to any other system on its own network. (In the case of Ethernet, it simply finds the Ethernet address of the destination system, and puts the datagram out on the Ethernet.) The problem comes when a system is asked to send a datagram to a system on a different network. This problem is handled by gateways. A gateway is a system that connects a network with one or more other networks. Gateways are often normal computers that happen to have more than one network interface. For example, we have a Unix machine that has two different Ethernet interfaces. Thus it is connected to networks 128.6.4 and 128.6.3. This machine can act as a gateway between those two networks. The software on that machine must be set up so that it will forward datagrams from one network to the other. That is, if a machine on network 128.6.4 sends a datagram to the gateway, and the datagram is addressed to a machine on network 128.6.3, the gateway will forward the datagram to the destination. Major communications centers often have gateways that connect a number of different networks. (In many cases, special-purpose gateway systems provide better performance or reliability than general-purpose systems acting as gateways. A number of vendors sell such systems.)

Routing in IP is based entirely upon the network number of the destination address. Each computer has a table of network numbers. For each network number, a gateway is listed. This is the gateway to be used to get to that network. Note that the gateway doesn't have to connect directly to the network. It just has to be the best place to go to get there. For example at Rutgers, our interface to NSFnet is at the John von Neuman Supercomputer Center (JvNC). Our connection to JvNC is via a high-speed serial line connected to a gateway whose address is 128.6.3.12. Systems on net 128.6.3 will list 128.6.3.12 as the gateway for many off-campus networks. However systems on net 128.6.4 will list 128.6.4.1 as the gateway to those same off-campus networks. 128.6.4.1 is the gateway between networks 128.6.4 and 128.6.3, so it is the first step in getting to JvNC.

When a computer wants to send a datagram, it first checks to see if the destination address is on the system's own local network. If so, the datagram can be sent directly. Otherwise, the system expects to find an entry for the network that the destination address is on. The datagram is sent to the gateway listed in that entry. This table can get quite big. For example, the Internet now includes several hundred individual networks. Thus various strategies have been developed to reduce the size of the routing table. One strategy is to depend upon "default routes". Often, there is only one gateway out of a network. This gateway might connect a local Ethernet to a campus-wide backbone network. In that case, we don't need to have a separate entry for every network in the world. We simply define that gateway as a "default". When no specific route is found for a datagram, the datagram is sent to the default gateway. A default gateway can even be used when there are several gateways on a network. There are provisions for gateways to send a message saying "I'm not the best gateway -- use this one instead." (The message is sent via ICMP. See RFC 792.) Most network software is designed to use these messages to add entries to their routing tables. Suppose network 128.6.4 has two gateways, 128.6.4.59 and 128.6.4.1. 128.6.4.59 leads to several other internal Rutgers networks. 128.6.4.1 leads indirectly to the NSFnet. Suppose we set 128.6.4.59 as a default gateway, and have no other routing table entries. Now what happens when we need to send a datagram to MIT? MIT is network 18. Since we have no entry for network 18, the datagram will be sent to the default, 128.6.4.59. As it happens, this gateway is the wrong one. So it will forward the datagram to 128.6.4.1. But it will also send back an error saying in effect: "to get to network 18, use 128.6.4.1". Our software will then add an entry to the routing table. Any future datagrams to MIT will then go directly to 128.6.4.1. (The error message is sent using the ICMP protocol. The message type is called "ICMP redirect.")

Most IP experts recommend that individual computers should not try to keep track of the entire network. Instead, they should start with default gateways, and let the gateways tell them the routes, as just described. However this doesn't say how the gateways should find out about the routes. The gateways can't depend upon this strategy. They have to have fairly complete routing tables. For this, some sort of routing protocol is needed. A routing protocol is simply a technique for the gateways to find each other, and keep up to date about the best way to get to every network. RFC 1009 contains a review of gateway design and routing. However rip.doc is probably a better introduction to the subject. It contains some tutorial material, and a detailed description of the most commonly-used routing protocol.

Details about Internet addresses: subnets and broadcasting

As indicated earlier, Internet addresses are 32-bit numbers, normally written as 4 octets (in decimal), e.g. 128.6.4.7. There are actually 3 different types of address. The problem is that the address has to indicate both the network and the host within the network. It was felt that eventually there would be lots of networks. Many of them would be small, but probably 24 bits would be needed to represent all the IP networks. It was also felt that some very big networks might need 24 bits to represent all of their hosts. This would seem to lead to 48 bit addresses. But the designers really wanted to use 32 bit addresses. So they adopted a kludge. The assumption is that most of the networks will be small. So they set up three different ranges of address. Addresses beginning with 1 to 126 use only the first octet for the network number. The other three octets are available for the host number. Thus 24 bits are available for hosts. These numbers are used for large networks. But there can only be 126 of these very big networks. The Arpanet is one, and there are a few large commercial networks. But few normal organizations get one of these "class A" addresses. For normal large organizations, "class B" addresses are used. Class B addresses use the first two octets for the network number. Thus network numbers are 128.1 through 191.254. (We avoid 0 and 255, for reasons that we see below. We also avoid addresses beginning with 127, because that is used by some systems for special purposes.) The last two octets are available for host addesses, giving 16 bits of host address. This allows for 64516 computers, which should be enough for most organizations. (It is possible to get more than one class B address, if you run out.) Finally, class C addresses use three octets, in the range 192.1.1 to 223.254.254. These allow only 254 hosts on each network, but there can be lots of these networks. Addresses above 223 are reserved for future use, as class D and E (which are currently not defined).

Many large organizations find it convenient to divide their network number into "subnets". For example, Rutgers has been assigned a class B address, 128.6. We find it convenient to use the third octet of the address to indicate which Ethernet a host is on. This division has no significance outside of Rutgers. A computer at another institution would treat all datagrams addressed to 128.6 the same way. They would not look at the third octet of the address. Thus computers outside Rutgers would not have different routes for 128.6.4 or 128.6.5. But inside Rutgers, we treat 128.6.4 and 128.6.5 as separate networks. In effect, gateways inside Rutgers have separate entries for each Rutgers subnet, whereas gateways outside Rutgers just have one entry for 128.6. Note that we could do exactly the same thing by using a separate class C address for each Ethernet. As far as Rutgers is concerned, it would be just as convenient for us to have a number of class C addresses. However using class C addresses would make things inconvenient for the rest of the world. Every institution that wanted to talk to us would have to have a separate entry for each one of our networks. If every institution did this, there would be far too many networks for any reasonable gateway to keep track of. By subdividing a class B network, we hide our internal structure from everyone else, and save them trouble. This subnet strategy requires special provisions in the network software. It is described in RFC 950.

0 and 255 have special meanings. 0 is reserved for machines that don't know their address. In certain circumstances it is possible for a machine not to know the number of the network it is on, or even its own host address. For example, 0.0.0.23 would be a machine that knew it was host number 23, but didn't know on what network.

255 is used for "broadcast". A broadcast is a message that you want every system on the network to see. Broadcasts are used in some situations where you don't know who to talk to. For example, suppose you need to look up a host name and get its Internet address. Sometimes you don't know the address of the nearest name server. In that case, you might send the request as a broadcast. There are also cases where a number of systems are interested in information. It is then less expensive to send a single broadcast than to send datagrams individually to each host that is interested in the information. In order to send a broadcast, you use an address that is made by using your network address, with all ones in the part of the address where the host number goes. For example, if you are on network 128.6.4, you would use 128.6.4.255 for broadcasts. How this is actually implemented depends upon the medium. It is not possible to send broadcasts on the Arpanet, or on point to point lines. However it is possible on an Ethernet. If you use an Ethernet address with all its bits on (all ones), every machine on the Ethernet is supposed to look at that datagram.

Although the official broadcast address for network 128.6.4 is now 128.6.4.255, there are some other addresses that may be treated as broadcasts by certain implementations. For convenience, the standard also allows 255.255.255.255 to be used. This refers to all hosts on the local network. It is often simpler to use 255.255.255.255 instead of finding out the network number for the local network and forming a broadcast address such as 128.6.4.255. In addition, certain older implementations may use 0 instead of 255 to form the broadcast address. Such implementations would use 128.6.4.0 instead of 128.6.4.255 as the broadcast address on network 128.6.4. Finally, certain older implementations may not understand about subnets. Thus they consider the network number to be 128.6. In that case, they will assume a broadcast address of 128.6.255.255 or 128.6.0.0. Until support for broadcasts is implemented properly, it can be a somewhat dangerous feature to use.

Because 0 and 255 are used for unknown and broadcast addresses, normal hosts should never be given addresses containing 0 or 255. Addresses should never begin with 0, 127, or any number above 223. Addresses violating these rules are sometimes referred to as "Martians", because of rumors that the Central University of Mars is using network 225.

Datagram fragmentation and reassembly

TCP/IP is designed for use with many different kinds of network. Unfortunately, network designers do not agree about how big packets can be. Ethernet packets can be 1500 octets long. Arpanet packets have a maximum of around 1000 octets. Some very fast networks have much larger packet sizes. At first, you might think that IP should simply settle on the smallest possible size. Unfortunately, this would cause serious performance problems. When transferring large files, big packets are far more efficient than small ones. So we want to be able to use the largest packet size possible. But we also want to be able to handle networks with small limits. There are two provisions for this. First, TCP has the ability to "negotiate" about datagram size. When a TCP connection first opens, both ends can send the maximum datagram size they can handle. The smaller of these numbers is used for the rest of the connection. This allows two implementations that can handle big datagrams to use them, but also lets them talk to implementations that can't handle them. However this doesn't completely solve the problem. The most serious problem is that the two ends don't necessarily know about all of the steps in between. For example, when sending data between Rutgers and Berkeley, it is likely that both computers will be on Ethernets. Thus they will both be prepared to handle 1500-octet datagrams. However the connection will at some point end up going over the Arpanet. It can't handle packets of that size. For this reason, there are provisions to split datagrams up into pieces. (This is referred to as "fragmentation".) The IP header contains fields indicating the a datagram has been split, and enough information to let the pieces be put back together. If a gateway connects an Ethernet to the Arpanet, it must be prepared to take 1500-octet Ethernet packets and split them into pieces that will fit on the Arpanet. Furthermore, every host implementation of TCP/IP must be prepared to accept pieces and put them back together. This is referred to as "reassembly".

TCP/IP implementations differ in the approach they take to deciding on datagram size. It is fairly common for implementations to use 576-byte datagrams whenever they can't verify that the entire path is able to handle larger packets. This rather conservative strategy is used because of the number of implementations with bugs in the code to reassemble fragments. Implementors often try to avoid ever having fragmentation occur. Different implementors take different approaches to deciding when it is safe to use large datagrams. Some use them only for the local network. Others will use them for any network on the same campus. 576 bytes is a "safe" size, which every implementation must support.

Ethernet encapsulation: ARP

There was a brief discussion earlier about what IP datagrams look like on an Ethernet. The discussion showed the Ethernet header and checksum. However it left one hole: It didn't say how to figure out what Ethernet address to use when you want to talk to a given Internet address. In fact, there is a separate protocol for this, called ARP ("address resolution protocol"). (Note by the way that ARP is not an IP protocol. That is, the ARP datagrams do not have IP headers.) Suppose you are on system 128.6.4.194 and you want to connect to system 128.6.4.7. Your system will first verify that 128.6.4.7 is on the same network, so it can talk directly via Ethernet. Then it will look up 128.6.4.7 in its ARP table, to see if it already knows the Ethernet address. If so, it will stick on an Ethernet header, and send the packet. But suppose this system is not in the ARP table. There is no way to send the packet, because you need the Ethernet address. So it uses the ARP protocol to send an ARP request. Essentially an ARP request says "I need the Ethernet address for 128.6.4.7". Every system listens to ARP requests. When a system sees an ARP request for itself, it is required to respond. So 128.6.4.7 will see the request, and will respond with an ARP reply saying in effect "128.6.4.7 is 8:0:20:1:56:34". (Recall that Ethernet addresses are 48 bits. This is 6 octets. Ethernet addresses are conventionally shown in hex, using the punctuation shown.) Your system will save this information in its ARP table, so future packets will go directly. Most systems treat the ARP table as a cache, and clear entries in it if they have not been used in a certain period of time.

Note by the way that ARP requests must be sent as "broadcasts". There is no way that an ARP request can be sent directly to the right system. After all, the whole reason for sending an ARP request is that you don't know the Ethernet address. So an Ethernet address of all ones is used, i.e. ff:ff:ff:ff:ff:ff. By convention, every machine on the Ethernet is required to pay attention to packets with this as an address. So every system sees every ARP requests. They all look to see whether the request is for their own address. If so, they respond. If not, they could just ignore it. (Some hosts will use ARP requests to update their knowledge about other hosts on the network, even if the request isn't for them.) Note that packets whose IP address indicates broadcast (e.g. 255.255.255.255 or 128.6.4.255) are also sent with an Ethernet address that is all ones.

Getting more information

This directory contains documents describing the major protocols. There are literally hundreds of documents, so we have chosen the ones that seem most important. Internet standards are called RFC's. RFC stands for Request for Comment. A proposed standard is initially issued as a proposal, and given an RFC number. When it is finally accepted, it is added to Official Internet Protocols, but it is still referred to by the RFC number. We have also included two IEN's. (IEN's used to be a separate classification for more informal documents. This classification no longer exists -- RFC's are now used for all official Internet documents, and a mailing list is used for more informal reports.) The convention is that whenever an RFC is revised, the revised version gets a new number. This is fine for most purposes, but it causes problems with two documents: Assigned Numbers and Official Internet Protocols. These documents are being revised all the time, so the RFC number keeps changing. You will have to look in rfc-index.txt to find the number of the latest edition. Anyone who is seriously interested in TCP/IP should read the RFC describing IP (791). RFC 1009 is also useful. It is a specification for gateways to be used by NSFnet. As such, it contains an overview of a lot of the TCP/IP technology. You should probably also read the description of at least one of the application protocols, just to get a feel for the way things work. Mail is probably a good one (821/822). TCP (793) is of course a very basic specification. However the spec is fairly complex, so you should only read this when you have the time and patience to think about it carefully. Fortunately, the author of the major RFC's (Jon Postel) is a very good writer. The TCP RFC is far easier to read than you would expect, given the complexity of what it is describing. You can look at the other RFC's as you become curious about their subject matter.

Here is a list of the documents you are more likely to want:

rfc-index	list of all RFC's
rfc1012	somewhat fuller list of all RFC's
rfc1011	Official Protocols. It's useful to scan this to see

what tasks protocols have been built for. This defines which RFC's are actual standards, as opposed to requests for comments.

rfc1010 Assigned Numbers. If you are working with TCP/IP, you will probably want a hardcopy of this as a reference. It's not very exciting to read. It lists all the offically defined well-known ports and lots of other things.

rfc1009 NSFnet gateway specifications. A good overview of IP routing and gateway technology.

rfc1001/2 netBIOS: networking for PC's

rfc973 update on domains

rfc959 FTP (file transfer)

rfc950 subnets

rfc937 POP2: protocol for reading mail on PC's

rfc894 how IP is to be put on Ethernet, see also rfc825

rfc882/3 domains (the database used to go from host names to Internet address and back -- also used to handle UUCP these days). See also rfc973

rfc854/5 telnet - protocol for remote logins

rfc826 ARP - protocol for finding out Ethernet addresses

rfc821/2 mail

rfc814 names and ports - general concepts behind well-known ports

rfc793 TCP

rfc792 ICMP

rfc791 IP

rfc768 UDP

rip.doc details of the most commonly-used routing protocol

ien-116 old name server (still needed by several kinds of system)

ien-48 the Catenet model, general description of the philosophy behind TCP/IP

The following documents are somewhat more specialized.

rfc813 window and acknowledgement strategies in TCP

rfc815 datagram reassembly techniques

rfc816 fault isolation and resolution techniques

rfc817	modularity and efficiency in implementation
rfc879	the maximum segment size option in TCP
rfc896	congestion control
rfc827,888,904,975,985	EGP and related issues

To those of you who may be reading this document remotely instead of at Rutgers: The most important RFC's have been collected into a three-volume set, the DDN Protocol Handbook. It is available from the DDN Network Information Center, SRI International, 333 Ravenswood Avenue, Menlo Park, California 94025 (telephone: 800-235-3155). You should be able to get them via anonymous FTP from sri-nic.arpa. File names are:

```
RFC's:
  rfc:rfc-index.txt
  rfc:rfcxxx.txt
IEN's:
  ien:ien-index.txt
  ien:ien-xxx.txt
```

rip.doc is available by anonymous FTP from topaz.rutgers.edu, as /pub/tcp-ip-docs/rip.doc.

Sites with access to UUCP but not FTP may be able to retreive them via UUCP from UUCP host rutgers. The file names would be

```
RFC's:
  /topaz/pub/pub/tcp-ip-docs/rfc-index.txt
  /topaz/pub/pub/tcp-ip-docs/rfcxxx.txt
IEN's:
  /topaz/pub/pub/tcp-ip-docs/ien-index.txt
  /topaz/pub/pub/tcp-ip-docs/ien-xxx.txt
/topaz/pub/pub/tcp-ip-docs/rip.doc
```

Note that SRI-NIC has the entire set of RFC's and IEN's, but rutgers and topaz have only those specifically mentioned above.

B Code examples

This is the code of the Gateway for the PC running Linux and the lnpd:

```
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/uio.h>
#include <sys/types.h>
#include "liblnp.h"
#define BUFSIZE 240
unsigned char send_buf[BUFSIZE];
unsigned char receive_buf[BUFSIZE];
static unsigned char sbuf_len;
static unsigned char rbuf_len;
static int fd;
/*---------------------------------------------------------------------------
----------*/
void
tundev_init(void)
{
  fd = open("/dev/tun0", O_RDWR);
  if(fd == -1) {
    perror("tun_dev: tundev_init: open");
    exit(1);
  }
#ifdef linux
  system("ifconfig tun0 inet 192.168.0.2 192.168.0.1");
  system("route add -net 192.168.0.0 netmask 255.255.255.0 dev tun0");
#else
  system("ifconfig tun0 inet 192.168.0.1 192.168.0.2");

#endif /* linux */
}
/*---------------------------------------------------------------------------
----------*/

void
tundev_send(void)
{
  int ret;
  ret = write(fd,receive_buf, rbuf_len);
  if(ret == -1) {
    perror("tun_dev: tundev_done: write");
    exit(1);
  }
}

/*---------------------------------------------------------------------------
----------*/
```

```
unsigned int
tundev_read(void)
{
  fd_set fdset;
  struct timeval tv;
  int ret;

  tv.tv_sec = 0;
  tv.tv_usec = 0;
  FD_ZERO(&fdset);
  FD_SET(fd, &fdset);

  ret = select(fd + 1, &fdset, NULL, NULL, &tv);
  if(ret == 0) {
    return 0;
  }
  ret = read(fd, send_buf, BUFSIZE);
  if(ret == -1) {
    perror("tun_dev: tundev_read: read");
  }

  return ret;
}
/*---------------------------------------------------------------------------
----------*/
void handler(const unsigned char* data,unsigned char length)
{
//read from RCX
//printf("got one\n");
memcpy(receive_buf,data,length);
rbuf_len=length;
        tundev_send(); //forward packet to ip-address
}
/*---------------------------------------------------------------------------
----------*/
```

```
int
main(void)
{

  lnp_tx_result result;

if ( lnp_init(0,0,0,0,0) )
    {
      perror("lnp_init");
      exit(1);
    }

    else fprintf(stderr,"init OK\n");


  tundev_init();

lnp_integrity_set_handler(handler);

  while(1) {
    sbuf_len = tundev_read(); //ready packet from ip-address
    if(sbuf_len == 0) {
    } else
     if(sbuf_len > 0) {
      result = lnp_integrity_write(send_buf,sbuf_len); //forward to RCX
      switch (result)
            {
            case TX_SUCCESS:
            sbuf_len=0;
            break;
            case TX_FAILURE:
            sbuf_len=0; // the packet is considered to be lost
                  break;
            default:
                  perror("Transmit error");
                  exit(1);
        }

     }

  }
  return 0;
}
/*---------------------------------------------------------------------------
----------*/
```

This is the code for the rcx:

```
#include <lnp.h>
#include <conio.h>
#include <sys/irq.h>
#include <sys/h8.h>
#include <tm.h>
#include <dkey.h>
#include <unistd.h>
#include <lnp-logical.h>
#include <sys/lnp-logical.h>
#include <uip.h>
#include <string.h>
#include <semaphore.h>
char done;

static char result;
static unsigned char packet_length;
static unsigned char tmpbuf[UIP_BUFSIZE];
static sem_t packet_sem; //synchronization semaphore

wakeup_t prgmKeyCheck(wakeup_t data) {
      return dkey == KEY_PRGM;
}

void prepareData() {
            u8_t i;
            for (i = 0; i < 40; i++) {
                  tmpbuf[i] = uip_buf[i];
            }
            for (i = 40; i < uip_len; i++) {
                  tmpbuf[i] = uip_appdata[i - 40];
            }
      }

/*---------------------------------------------------------------------------
----------*/

      /*
      The packet handler gets all integrity packets sent over the network.
      The handler sets uip_len to length
      (everytime a packet arrives, its passed to this handler)
      */

      void packet_handler(const unsigned char * data, unsigned char length)
{
             if(packet_length>0 || length==0)
            return;
            memcpy(uip_buf, data,length);
            uip_len = length; // set new length of packet
            packet_length = length;
            sem_post(&packet_sem);
      }

/*---------------------------------------------------------------------------
----------*/
```

```
int sender(int argc, char* argv[]) {
      while (!done) {
            // wait for new packet
            packet_length = 0;
            sem_wait(&packet_sem);
            if (uip_len > 0) {
                  uip_process(UIP_DATA);
                  if (uip_len > 0) {
                        prepareData();
                        result = lnp_integrity_write(tmpbuf, uip_len);
                        if (result == TX_IDLE) {
                              msleep(10);
                        }
                        uip_len = 0;
                  }
            }
      }
      return 0;
}
/*----------------------------------------------------------------------------
----------*/
int main(int argc, char** argv) {
            uip_init();
            lnp_logical_range(0);
            lcd_clear();
            uip_len = 0;
            lnp_integrity_set_handler(packet_handler);
            packet_length = 0;
            sem_init(&packet_sem,0,0);
            execi(sender, 0, NULL, PRIO_NORMAL, DEFAULT_STACK_SIZE);
            while (1) {
                  if (prgmKeyCheck(0))
                        break;
                  // this program must be terminated by the prgm key !!!
            }
            done = 1;
            cputs("done");
            sem_destroy(&packet_sem);
            return 0;
            // we re all done
      }
/*----------------------------------------------------------------------------
----------*/
```

The following code is a modified version of the httpd that comes with the original uIP distribution.
For further information please take a look at the sources of the uIP distribution available from Adam Dunkels website.
Own code for the uIP on the RCX should be written as a replacement of the httpd sample application and almost exactly the way this application is written.

```
/*
 * Copyright (c) 2001, Adam Dunkels.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *      This product includes software developed by Adam Dunkels.
 * 4. The name of the author may not be used to endorse or promote
 *    products derived from this software without specific prior
 *    written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS
 * OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY
 * DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
 * GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * This file is part of the uIP TCP/IP stack.
 *
 * $Id: httpd.c,v 1.4 2002/03/10 19:27:42 adam Exp $
 *
 */


#include "uip.h"
#include "httpd.h"
#include "fs.h"
#include "fsdata.h"
#include "cgi.h"

#include <dmotor.h>

#define NULL (void *)0

/* The HTTP server states: */
#define HTTP_NOGET        0
#define HTTP_FILE         1
#define HTTP_TEXT         2
#define HTTP_FUNC         3
#define HTTP_END          4

#define PRINT(x)
```

```
#define PRINTLN(x)

struct httpd_state *hs;

extern const struct fsdata_file file_index_html;

static void next_scriptline(void);
static void next_scriptstate(void);

#define ISO_G        0x47
#define ISO_E        0x45
#define ISO_T        0x54
#define ISO_slash    0x2f
#define ISO_c        0x63
#define ISO_g        0x67
#define ISO_i        0x69
#define ISO_space    0x20
#define ISO_nl       0x0a
#define ISO_cr       0x0d
#define ISO_a        0x61
#define ISO_t        0x74
#define ISO_hash     0x23
#define ISO_period   0x2e

extern u8_t motor_a_run, motor_c_run;

/*-----------------------------------------------------------------------------------*/
void
httpd_init(void)
{
  fs_init();

  /* Set up a web server. */
  uip_listen(80);

  uip_listen(6510);
}
/*-----------------------------------------------------------------------------------*/
void
httpd(void)
{
  struct fs_file fsfile;
  u8_t i;

  hs = (struct httpd_state *)(uip_conn->appstate);

  switch(uip_conn->lport) {
  case htons(80):
    cputs("http ");
    uip_len = 0;
    /* We use the uip_flags variable to deduce why we were called. If
       uip_flags & UIP_ACCEPT is non-zero, we were called because a
       remote host has connected to us. If uip_flags & UIP_NEWDATA is
       non-zero, we were called because the remote host has sent us
       new data, and if uip_flags & UIP_ACKDATA is non-zero, the
       remote host has acknowledged the data we previously sent to
       it. */
    if(uip_flags & UIP_ACCEPT) {
      /* Since we have just been connected with the remote host, we
             reset the state for this connection. The ->count variable
             contains the amount of data that is yet to be sent to the
             remote host, and the ->state is set to HTTP_NOGET to signal
```

```
         that we haven't received any HTTP GET request for this
         connection yet. */
  hs->state = HTTP_NOGET;
  hs->count = 0;
  uip_len = 0;
  return;

} else if(uip_flags & UIP_POLL) {
  /* If we are polled ten times, we abort the connection. This is
         because we don't want connections lingering indefinately in
         the system. */
  if(hs->count++ >= 10) {
        uip_flags = UIP_ABORT;
  }
  return;
} else if(uip_flags & UIP_NEWDATA && hs->state == HTTP_NOGET) {
  /* This is the first data we receive, and it should contain a
         GET. */

  /* Check for GET. */
  if(uip_appdata[0] != ISO_G ||
         uip_appdata[1] != ISO_E ||
         uip_appdata[2] != ISO_T ||
         uip_appdata[3] != ISO_space) {
        uip_flags = UIP_ABORT;
        return;
  }

  /* Find the file we are looking for. */
  for(i = 4; i < 40; ++i) {
        if(uip_appdata[i] == ISO_space ||
           uip_appdata[i] == ISO_cr ||
           uip_appdata[i] == ISO_nl) {
          uip_appdata[i] = 0;
          break;
        }
  }

  PRINT("request for file ");
  PRINTLN(&uip_appdata[4]);

  if(!fs_open((const char *)&uip_appdata[4], &fsfile)) {
        PRINTLN("couldn't open file");
        fs_open(file_index_html.name, &fsfile);
  }

  if(uip_appdata[4] == ISO_slash &&
         uip_appdata[5] == ISO_c &&
         uip_appdata[6] == ISO_g &&
         uip_appdata[7] == ISO_i &&
         uip_appdata[8] == ISO_slash) {
        /* If the request is for a file that starts with "/cgi/", we
           prepare for invoking a script. */
        hs->script = fsfile.data;
        next_scriptstate();
  } else {
        hs->script = NULL;
        /* The web server is now no longer in the HTTP_NOGET state, but
           in the HTTP_FILE state since is has now got the GET from
           the client and will start transmitting the file. */
        hs->state = HTTP_FILE;
        /* Point the file pointers in the connection state to point to
           the first byte of the file. */
        hs->dataptr = fsfile.data;
        hs->count = fsfile.len;
  }
}


if(hs->state != HTTP_FUNC) {
  /* Check if the client (remote end) has acknowledged any data that
         we've previously sent. If so, we move the file pointer further
         into the file and send back more data. If we are out of data to
         send, we set the UIP_CLOSE flag in the uip_flags variable to
         tell uIP that the connection should be closed. */
  if(uip_flags & UIP_ACKDATA) {

        if(hs->count >= UIP_TCP_MSS) {
          hs->count -= UIP_TCP_MSS;
          hs->dataptr += UIP_TCP_MSS;
        } else {
          hs->count = 0;
        }

        if(hs->count == 0) {
          if(hs->script != NULL) {
            next_scriptline();
            next_scriptstate();
          } else {
            uip_flags = UIP_CLOSE;
          }
        }
  }
}


if(hs->state == HTTP_FUNC) {

  /* Call the CGI function. */
  if(cgitab[hs->script[2] - ISO_a]()) {
        /* If the function returns non-zero, we jump to the next line
           in the script. */
        next_scriptline();
        next_scriptstate();
  }
}

if(hs->state != HTTP_FUNC && !(uip_flags & UIP_POLL)) {
  /* The uip_len variable should contain the length of the
         outbound packet. */
  if(hs->count > UIP_TCP_MSS) {
        uip_len = UIP_TCP_MSS;
  } else {
        uip_len = hs->count;
  }

  /* Set the uip_appdata pointer to point to the data we wish to
         send. */
  uip_appdata = hs->dataptr;
}

/* Finally, return to uIP. Our outgoing packet will soon be on its
   way... */
```

```
    return;

#define MOVE_RIGHT 1
#define MOVE_LEFT  2
#define MOVE_UP    3
#define MOVE_DOWN  4

  case htons(6510):
    if(uip_flags & UIP_ACCEPT) {
      hs->count = 0;
    }
    if(uip_flags & UIP_POLL) {
      if(++hs->count >= 60) {
              uip_flags = UIP_ABORT;
      }
    }
    if(uip_flags & UIP_NEWDATA) {
      hs->count = 0;
      while(uip_len > 1) {
        switch(*uip_appdata) {
        case MOVE_RIGHT:
                motor_a_dir(fwd);
                motor_a_speed(MAX_SPEED);
                motor_a_run = uip_appdata[1];
          break;
        case MOVE_LEFT:
                motor_a_dir(rev);
                motor_a_speed(MAX_SPEED);
                motor_a_run = uip_appdata[1];
          break;
        case MOVE_UP:
                motor_c_dir(fwd);
                motor_c_speed(MAX_SPEED);
                motor_c_run = uip_appdata[1];
          break;
        case MOVE_DOWN:
                motor_c_dir(rev);
                motor_c_speed(MAX_SPEED);
                motor_c_run = uip_appdata[1];
          break;
        }
        ++uip_appdata;
        --uip_len;
        ++uip_appdata;
        --uip_len;
      }
    }
    break;
  }
}
/*-----------------------------------------------------------------------------------*/
static void
next_scriptline(void)
{
  /* Loop until we find a newline character. */
  do {
    ++(hs->script);
  } while(hs->script[0] != ISO_nl);

  /* Eat up the newline as well. */
  ++(hs->script);
}
```

```
/*-----------------------------------------------------------------------------------*/
static void
next_scriptstate(void)
{
  struct fs_file fsfile;
  u8_t i;

 again:
  switch(hs->script[0]) {
  case ISO_t:
    /* Send a text string. */
    hs->state = HTTP_TEXT;
    hs->dataptr = &hs->script[2];

    /* Calculate length of string. */
    for(i = 0; hs->dataptr[i] != ISO_nl; ++i);
    hs->count = i;
    break;
  case ISO_c:
    /* Call a function. */
    hs->state = HTTP_FUNC;
    hs->dataptr = NULL;
    hs->count = 0;
    uip_flags = 0;
    break;
  case ISO_i:
    /* Include a file. */
    hs->state = HTTP_FILE;
    if(!fs_open(&hs->script[2], &fsfile)) {
      uip_flags = UIP_ABORT;
    }
    hs->dataptr = fsfile.data;
    hs->count = fsfile.len;
    break;
  case ISO_hash:
    /* Comment line. */
    next_scriptline();
    goto again;
    break;
  case ISO_period:
    /* End of script. */
    hs->state = HTTP_END;
    uip_flags = UIP_CLOSE;
    break;
  default:
    uip_flags = UIP_ABORT;
    break;
  }
}
/*-----------------------------------------------------------------------------------*/
```

Glossary

ACK
The acknowledgment signal used by TCP.
Addressing layer
The LNP layer, that uses addressing packets for transmission.
Addressing packet
Packet, holding addressing information
checksum
A checksum is a function that computes a speci¯c number by summing all bytes in a packet. Used for detection of data corruption.
congestion
Congestion occurs when a router drops packets due to full bu®ers, i.e., when the network is overloaded.
datagram
A chunk of information. Analogous to a packet.
demultiplexing
The opposite of multiplexing. Extracting one of the information streams from a combined stream of information streams.
header
Control information for a packet located at the beginning of the packet.
ICMP
Internet Control Message Protocol. An unreliable signaling protocol used together with IP.
internet
An interconnected set of networks using IP for addressing.
Internet
The global Internet.
Integrity layer
The LNP layer, that uses integrity packets for transmission.
Integrity packet
Packet, without any addressing information
IP
Internet Protocol. The protocol used for addressing packets in an internet.
IPv4
Internet Protocol version 4. The version of IP mainly used in the global Internet.
IPv6
Internet Protocol version 6. The next generation IP. Expands the address space from 2 32 combinations to 2 128 and also supports auto-conguration.
LegOS
Alternative operatingssytem for the RCX..
LNP
Layered network protocol. LNP is the network protocol developed for the RCX.
lnpd
A Linux deamon, written by Martin Cornelius.
Mindstorm
The other name of the RCX.
multiplexing
A technique that enables two or more information streams to use the same link. In the TCP/IP case this refers to the process of, e.g., using the IP layer for many di®erent protocols such as UDP or TCP.
packet
A chunk of information. Analogous to a datagram.
PCB
Protocol Control Block. The data structure holding state related information of a (possibly half-open) UDP or TCP connection.
proxy
An intermediate agent that utilizes knowledge of the transportation mechanism to enhance performance.

RFC
Request For Comments. A paper specifying a standard or discussing various mechanisms in the Internet.
round-trip time
The total time it takes for a packet to travel from the sender to the receiver, and for the reply to travel back to the sender.
router
A node in an internet. Connects two or more networks and forwards IP packets across the connection point.
UDP
User Datagram Protocol. An unreliable datagram protocol on top of IP. Mainly used for delay sensitive applications such as real time audio and video.
uIP
a very tiny TCP/IP-stack, written by Adam Dunkels.
TCP
Transmission Control Protocol. Provides a reliable byte stream on top of IP. The most commonly used transportation protocol in todays Internet. Used for email as well as ¯le and web services.
TCP/IP
The internet protocol suite which includes the basic delivery protocols such as IP, UDP and TCP as well as some application level protocols such as the email transfer protocol SMTP and the ¯le transfer protocol FTP.
Transceiver
The transmitter /receiver of the RCX.
Wireless Network
A network that uses electromagnetic waves such as infra red or radio to transmit data.

Bibliography

Adam Dunkels
http://dunkels.com/adam/
LwIP (November 2000) and uIP (June 2001)

[BIG + 97] C. Brian, P. Indra, W. Geun, J. Prescott, and T. Sakai. IEEE-
802.11 wireless local area networks. *IEEE Communications Magazine*, 35(9):116{126, September 1997.

[**COR2000**] Martin Cornelius
Lnpd, a Linux daemon allowing multiple clients to connect to a LEGO RCX running legOS

[HNI + 98] J. Haartsen, M. Naghshineh, J. Inouye, O. Joeressen, and W. Allen.
Bluetooth: Vision, goals, and architecture. *Mobile Computing and Communications Review*, 2(4):38{45, October 1998.

[NIE2000]
Introduction to the legOS kernel.
Stig Nielsson
September 2000
http://legos.sourceforge.net/docs/kerneldoc.ps

TCP and UDP Performance over a Wireless LAN
George Xylomenos and George C. Polyzos
Center for WirelessCommunications & Computer Systems Laboratory.
University of California, San Diego.

TCP Performance over Wireless Links
Amol Shah
EE359 – Wireless Communications
Autumn 2001
Stanford University
December 12, 2001

Performance Evaluation of TCP Over Wireless Links
Praveen Sheethalnath Shalaka Tendulkar
Sonal Ramnani Laurent Rival

Introduction to the Internet Protocols
Computer Science Facilities Group
RUTGERS
The State University of New Jersey
3 July 1987

PiggyData:
Reducing CSMA/CA collisions for multimedia and TCP connections:
*Jean Tourrilhes(*jt@hplb.hpl.hp.com)
Hewlett Packard Laboratories, Filton Road, Bristol BS34 6QZ, U.K
May 99

Network Magazine
http://www.networkmagazineindia.com/200102/focus1.htm
February 2001

Kekoa Proudfoot
http://graphics.stanford.edu/~kekoa/
RCX Internals
1998,1999