Amateur TCP/IP in 1989

Phil Karn, KA9Q

ABSTRACT

This paper is a report on the status of the KA9Q Internet Protocol package, also known as NET. Most of the items proposed in last year's paper have been completed, and additional features have also been implemented by the author and other contributors. The use of TCP/IP with the WA4DSY 56kb/s modems is also discussed, along with some ideas on channel access methods that would improve the efficiency of these modems.

Introduction

Amateur TCP/IP continues to grow in popularity. In ampr.org, the portion of the ARPA Internet name space reserved for amateur packet radio, there are now over 2300 registered IP address assignments in 29 countries. NET/ROM networks as well as ''plain'' AX.25 paths are being used to carry IP datagrams. Most of this activity uses convention 1200 baud modems, but higher speed operation with WA4DSY 56 kbps modems is also growing. A newsletter edited by Rich Vitello WA1EQU, ''The New England TCPer'', is devoted to amateur TCP/IP operation.

NOS — Software Restructuring

I have completed the internal restructuring of the NET software described in last year's paper. The original version, on which the ''Dayton'' releases 890421.0 and .1 from N3EUA are based, used a commutator loop structure. External events such as packet arrivals and protocol state changes, made ''upcalls'' to functions within each application that did the actual work. Applications therefore had to be structured as state machines driven by external events (the upcalls), and this became very clumsy for the more complex applications such as the FTP client.

The new version, referred to as ''NOS'', is internally quite different as it supports internal multitasking. Each task is a logically separate program. Tasks have private stacks; in C, that means they have their own automatic variables. On the other hand, they share the same set of static and global variables. Tasks may block, that is, they can call a special system function that does not return until a specified event happens. Whenever a task blocks, it is ''put to sleep'' and the system picks another to run until it too blocks. Tasks may wait for events that are signaled by hardware interrupts (e.g., when a packet arrives or a key is pressed) or software (e.g., messages from other tasks.)

The kernel (that part of the system that controls the running of tasks) is non preempting. That is, when a task gains control of the system, it continues to run until it explicitly relinquishes control by waiting for an event. This is in contrast to the preempting kernel commonly found in timesharing systems, where a running task can be suspended and another run at any time by an interrupt. The choice of a non preempting kernel is an important feature, since it avoids a whole class of potential bugs in which shared data being modified by one task is left in an inconsistent state when preemption occurs. The only situation that requires care is the modification of global data by an interrupt handler, since interrupts can happen at any time. There are only a few such situations within NOS, and the tasks avoid conflicts by temporarily disabling interrupts whenever the shared data is examined or modified.

The main drawback to the non-preempting approach is poorer real-time response; even if an external event (e.g., a packet arriving) makes a task runnable, it will not get control until the currently running task gets around to giving up the system and any other runnable tasks have had their turns. In a networking system like NET, the only significant ''real time'' requirement is that incoming data not be lost, so incoming data is buffered at interrupt time until the main network task can get a chance to process it. This only requires that hardware interrupts not be disabled for excessive periods and that the network task gain control often enough to empty the incoming buffers before they overflow. However, the tasks that make up NET seldom run for more than a few milliseconds at a time before blocking, so this is not much of a concern. With the exception of the ‘‘hs’’ HDLC driver discussed later, the only significant latencies occur when a task reads or writes a block in the MS-DOS file system. Because the MS-DOS file system is synchronous, NET cannot run during disk I/O. Fixing this would require rewriting substantial portions of MS-DOS and the ROM BIOS, and is outside the scope of the project.

The ‘‘memory model’’ used by NET on the IBM PC also changed during the conversion to NOS. For some time, the ‘‘medium model’’ (large code, small data) was used; now the ‘‘large model’’ (large code and data) is used, allowing the use of more than 64K for data buffering. Although the need for more data space rose gradually as the package grew, the main impetus for change was the need to allocate a stack for each task. The conversion to the large data model was much easier than I had expected. The only module affected was the storage allocator, the only portion of the package that deals with data blocks larger than 64K. The enlargement of block size fields From 16 to 32 bits and the addition of the ‘‘huge’’ keyword to various pointers was the only change required to the allocator.

One problem that is still present in NOS is the need to keep careful track of allocated storage blocks. Unlike a conventional time sharing kernel (e.g., UNIX), storage allocated by a task in NOS is not automatically freed when the task completes. Tasks frequently allocate storage (e.g., a data buffer) and immediately pass it off to another task, and I considered it too time-consuming to have to keep careful ownership records for each allocated block. Dynamically allocated storage is also used in global data structures (e.g., TCP control blocks) that do not necessarily belong to any particular task, and this would complicate the record keeping even more.

Shortly after creating NOS I decided to switch to the Borland Turbo-C compiler as the primary compiler for the PC version. This change was made for several reasons: the wider availability and lower cost of Turbo-C, making it easier for others to configure and modify the code; fewer compiler bugs, especially in the new ANSI C features; better support for PC-specific hardware functions, allowing the elimination of some assembler code from the package; and the provision of an assembler with a built-in macro allowing C-callable assembler functions to access their arguments in a memory-model independent fashion.

Sockets

Very little was changed inside the protocol modules per se in the NOS conversion. A new module was instead placed on top of the existing TCP and UDP modules, implementing an application programming interface based closely on the ‘‘socket’’ mechanism in the Berkeley version of UNIX. The socket code provides the upcalls required by the existing TCP and UDP code . These upcall functions signal events internal to the socket module so that the socket primitives (e.g., send or receive data) can block appropriately. Applications using the socket interface do not need to provide any upcall functions.

The Berkeley socket interface supports protocols besides those of the Internet. It was therefore fairly easy to allow applications to access the AX.25 and NET/ROM protocols by creating two new ‘‘address classes’’, AF_AX25 and AF_NETROM. Either the virtual-circuit mode (using I frames) or the datagram mode (using UI frames) can be selected.

Although the socket interface follows the Berkeley UNIX model as closely as possible, there are some unavoidable differences. The most important difference is that socket descriptors in NOS are distinct from file descriptors, unlike Berkeley UNIX where they occupy a common set of integers. In other words, socket descriptor 1 does not refer to file descriptor 1, and the regular file I/O functions read() and write() cannot be used on sockets. All socket I/O must be through the recv(), send() and related functions. There is also a special close_s() call for sockets, since the regular close() function expects a file descriptor. Emulating the Berkeley model more closely would require either a major internal change to Borland’s standard C library or an independent reimplementation, and these changes could not be applied very easily to non MS-DOS environments.

All of the applications in NET have been rewritten to use sockets. In many cases the code became considerably simpler and smaller, particularly in the more complex applications such as the FTP client. The simpler programming style made it possible to add features to the clients relatively easily; for example, the FTP client now prompts for user names and passwords during the login sequence, and it automatically suppresses echoing of the password as it is typed in. The user may now ‘‘type ahead’’ a series of commands to FTP without having to wait for each transfer to complete first.

Domain Name Service

Another feature that was added after the conversion to NOS is domain name support. The original version of NET relied on a local table of host names (stored in the file /hosts.net) to translate machine names (e.g., ka9q.ampr.org) to numerical IP addresses. This technique was originally used by all of the systems on the ARPA Internet, but as the net grew this method rapidly began showing signs of strain.

The standard method now used in the Internet for name-to-address translation is the ‘‘domain name system’’ that consists of a distributed hierarchical database and servers that provide access to this data. Names in the domain name system are hierarchical, with the top level on the right. Levels in the hierarchy are separated by periods, and the system allows delegation of naming authority to a different entity at each level. It is important to understand that a domain name does not say *anything* about the geographic location of the system being named, and it is even more important to understand that *the individual components of a domain name do not constitute a route to that system*. The only purpose of the domain system is the hierarchical division of the name space along administrative or political lines; routing is an entirely separate function that belongs down in the network layer.

The NOS version includes a domain client only; that is, it requires Internet access to a server running on a different type of system in order to work automatically. However, responses from the servers are stored locally in the file /domain.txt and reused, and the user may add entries to this file manually if a domain server is not available via the net.

The domain name resolver is a good example of a feature that was fairly easy to add to NOS but would have been quite difficult to add to the old version. Applications querying the domain system must wait for a response. Doing this in the old version would have required additional states in the application representing domain server response waits, while in NOS the change to an application was trivial because the resolve function blocks automatically.

High speed TCP/IP

Several groups have been using the WA4DSY 56kb/s modem with TCP/IP. Two methods of interfacing the modem to the host computer are being used. The first is the modified KISS TNC as pioneered by the GRAPES group in Atlanta. The PC-to-TNC link runs at 19.2 kb/s, so a single station is unable to use the full channel bandwidth.

The other approach uses the plug-in HDLC cards that have become available for the IBM PC over the past few years. Richard Bisbey, NG6Q, and Art Goldman, WA3CVG, have written a driver for the ‘‘Eagle’’ card, a surplus 8530 adapter card. Their driver makes use of the Eagle card’s DMA feature, but unfortunately DMA is not supported on all of the newer HDLC cards such as the Digital Radio Systems (DRSI) PCPA. I have therefore written the ‘‘hs’’ driver that uses the PCPA (or Eagle card) in a programmed I/O mode; that is, the host CPU copies each byte to or from the device as required instead of relying on DMA hardware. Both drivers work, but mine requires an essentially dedicated system; whenever the modem is active the system ‘‘locks up’’ and is unable to do anything else but service the modem.

The ‘‘Awesome I/O’’ interface card designed by K3MC represents the best way to interface a fast modem to the IBM PC. It can buffer individual characters on its own, so the host PC need deal only with complete packets, which have much looser real time requirements. When the I/O card services the modem the host PC is free to respond to other interrupts. This will effectively eliminate the need to dedicate a PC to handle the modem, and it will allow multiple modems to be handled on the same system.

The network configurations at KA9Q and WB0MPQ consists of ‘‘stripped’’ PC/XTs that operate as dedicated IP routers (packet switches) between local Ethernets and the 56kb/s packet radio channel on 220.55 MHz. The theoretical throughput for a file transfer over this path is about 5700 bytes/sec, assuming a data packet size of 1400 bytes, a modem keyup delay of 15 ms, 56 bytes of TCP/IP/AX.25 protocol overhead, and ‘‘stop and wait’’ operation of TCP. It is interesting to note that the 15 ms modem keyup delay accounts for about twice as much overhead as the three protocol headers combined.

However, the actual throughput between PC/ATs is only about 3200 bytes/sec. There are two reasons for this. A minor cause is occasional TCP retransmission caused by link noise, but the major degradation is caused by the inability to overlap packet transmission or reception at each site with disk I/O due to the stop-and-wait mode in which we operate TCP. This latter reason is the main one, but we have found that stop-and-wait mode is necessary to prevent collisions between data packets and their acknowledgements on the radio channel that would degrade throughput much more severely.

Collisions — Again

These collisions can occur even without hidden terminals on the channel because the modems take a finite amount of time to key up or to recognize that another modem has keyed up, and these delays create ‘‘collision windows’’. Fixing this problem (which occurs on 1200 baud channels too) is a major challenge to amateur packet radio, and I have been investigating two possible solutions to the problem.

The first approach is collision detection, the theory being that collisions aren’t so bad if you can detect them quickly enough to abort a transmission before it has wasted much channel time. The colliding transmitters then delay for random intervals and again attempt to transmit their packets. If another collision occurs, the transmitters repeat the procedure with random delays chosen from ever-increasing intervals.

This is the approach used with Ethernet, also known as IEEE 802.3. Ethernet has been shown to be very stable even under pathological overload because of its collision detection feature. Useful throughput can be well above 90%, depending on the length of the cable and the size of the packets, with the better efficiencies coming from shorter cables and longer packets.

Collision detection is fairly easy to do on coax because of the relatively low attenuation. The much larger difference between transmitted and received signal levels in radio effectively makes collision detection impossible on a simplex radio channel. However, collision detection is possible through a repeater. If a user station operates in full duplex, it can compare its own signal heard through the repeater with the transmitted data. If no errors are seen, then the transmitting station can feel confident that the packet was not interfered with.

Implementing collision detection with the WA4DSY modems requires a repeater that can pass the wideband signal the modem produces with a minimum of distortion. One way to do this is with a linear translator similar to the transponders carried on the AMSAT communication satellites. A translator bandwidth of 100 KHz would be enough to accommodate the 75 KHz wide 56kb/s signal with guard bands on each side to avoid the phase distortion at the edges of the translator’s bandpass filter. As with a satellite transponder, this translator should operate in a crossband mode in order to make full duplex operation at the user stations relatively simple.

There is another approach to the collision problem that also deserves investigation: token passing. This technique requires considerably more complex software algorithms than collision detection, but it has the advantage of being usable on a simplex radio channel with no extra hardware. The IEEE 802.4 token-passing bus standard may be useful as a starting point, though a radio channel would be much more complex than the wire bus because of hidden terminals and higher noise levels. The model here is that of a voice round table; stations pass the token (the right to transmit) around a list of stations, with periodic pauses to allow new stations (‘‘breakers’’) to join the logical ring. To minimize the overhead spent on token passing, stations should remove themselves from the logical ring when they do not expect to send data for a while.

Acknowledgements

Many people have made contributions to the TCP/IP project, but I would like to cite several for their major contributions in the past year: Dan Frank W9NK, Anders Klemets SM0RGV, Russ Nelson, Dewayne Hendricks WA8DZP and Bdale Garbee N3EUA.

Dan has added a NET/ROM transport layer module to his earlier network layer code. Anders ported Dan’s code over to the NOS version, adding socket level code to support both NET/ROM and AX.25 connections. Russ has assembled a significant collection of packet drivers for quite a few Ethernet controllers; these drivers can be used by any protocol software that supports the FTP Software Packet Driver specification, including the KA9Q package. Dewayne has put a considerable amount of work into porting the NET software to the Apple Macintosh, giving it a Mac-like user interface. And Bdale Garbee has done a fantastic job in packaging releases of the pre-NOS PC code for general amateur release and in answering user questions.