Network Support under VxWorks

Strategies

Mon, Dec 14, 1998

The VxWorks operating system kernel provides for extensive networking support. Part of this support is the network sockets interface for the TCP/IP suite of Internet protocols. This note discusses strategies for using the sockets interface network support for the IRM-style system software.

The communications support needed by the system code is based upon UDP. The socket interface supports UDP transactions via the `sendto` and `recvfrom` functions. The calling sequences are as follows:

```
sendto(int s, caddr_t buf, int bufLen, int flags,
            struct sockaddr *to, int tolen)

recvfrom(int s, char *buf, int bufLen, int flags,
            struct sockaddr *from, int *pFromLen)
```

`s` = the socket used to send a datagram to or receive a datagram from.
`buf` = the buffer containing the datagram to be sent or received.
`bufLen` = length of datagram buffer
`flags` = flags to underlying protocols
`to` = buffer holding recipient's socket address
`from` = buffer for holding sender's socket address
`tolen` = length of <`to`> socket address
`pFromLen` = value/result of <`from`>

The socket number is returned from invoking the function

```
socket(int domain, int type, int protocol)
```

`domain` = protocol family (e.g. PF_INET = 2)
`type` = SOCK_DGRAM
`protocol` = socket protocol (usually 0)

A socket address is housed in a `sockaddr_in` structure.

```
struct sockaddr_in {
      u_short sin_family;  /* address family AF_INET = 2 */
      u_short sin_port;  /* protocol port# */
      u_long sin_addr;  /* IP address */
      char sin_zero[8];  /* unused (set to zero) */
      }
```

The socket address layout is simply a structure for holding an IP address and a port#. It is normally declared as a structure 16-bytes in length. Note that when using `recvfrom`, a pointer to the length of this structure is passed. The variable must be initialized to be large enough for the entire structure that is provided. On return the variable will be set to the size of the structure actually used to return the socket address value. One might set the variable that holds the socket address length to 16, then find that it holds the value 8 upon return from `recvfrom`.

It is important to note that the `sendto` function returns only after the datagram has been delivered to the network. The return value is the number of bytes transmitted, or ERROR (–1).

A significant part of system support for networking is that which combines multiple messages into a single datagram, when they are queued sequentially and do not exceed the maximum datagram size. The system operates at the message level, and it seeks to queue up messages that are to be sent to the network. It needs to keep things moving, because it is trying to operate in real-time (15 Hz). Therefore, it continues to queue messages to other nodes without waiting for transmissions to complete to a given node. But the way `sendto` operates in the socket interface, this cannot easily happen, as each `sendto` call awaits completion of the transmission. And in the case that the target node is not present in the local node's ARP table, an ARP request will have to be sent, and an ARP reply received, before the transmission can be initiated.

The front end is mostly a server; it supplies data in response to host requests. It operates at 15 Hz, so that at one point in the 15 Hz cycle, the system builds replies to all outstanding data requests for delivery to the various requesting nodes. The linked list of data requests is maintained in an order that is sorted by target node, so as to increase the likelihood that multiple messages destined for the same target node will be queued consecutively, which in turn allows coalescing of multiple messages that are destined for the same target node into a common datagram. So that all reply messages do not have to be queued before the first datagram is sent to its target node, a check is made for each message processed. If the target node is different from the oldest message queued, the queue is flushed to the network.

To get around the problem that `sendto` will not return until it is finished, we can utilize a high-priority task whose job it is to transmit datagrams; i.e., this task will be the one that invokes `sendto`. A task that flushes the network queue, which results in one or more datagrams being constructed each from one or more messages, sends a short reference message to the message queue on which the high-priority transmission task awaits. It awakes and calls `sendto`, which will block until the transmission is complete. Upon return, the review of transmitted messages to mark their completion in the network queue can be performed. (This logic is done currently by the network transmit interrupt code.) During the time that `sendto` blocks, the original task can continue building additional datagrams for transmission in the same way. The high priority task will keep the network hardware busy.

A wrinkle in the above scheme shows up in the case that the target node is not in the ARP table. The function `etherAddrResolve` may be used to assist with this.

```
etherAddrResolve(struct ifnet *pIf, char *targetAddr, char *eHdr,
int numTries, int numTicks)
```

This routine sends an ARP request if needed, else it signals that it already has the hardware address. If it sent an ARP request, it may be some time before the ARP reply is received. To handle this, we may use another high-priority task that handles transmissions to target nodes for which an ARP request was sent as a result of invoking

`etherAddrResolve`. Thus we will have two high-priority tasks that invoke `sendto`. Their code can be the same, except that they await different message queues. In this way, target nodes that need ARP handling will not impede transmissions to target nodes that do not. In summary, the code that is to send a datagram to a target node first invokes `etherAddrResolve`, then writes a reference message to one of two message queues. A higher-priority task will see to it that `sendto` is invoked as necessary to get the datagrams delivered to the network.

The reason all the above is necessary is that the socket interface is not designed to operate in real-time. It originated with unix. It has no buffered output operations.

Multicast support

VxWorks supplies support for IP multicasting via its SENS enhancement package. (This multicasting support may be included in a future release of VxWorks.)

The IRM system software needs to know whether a message it receives was multicast or unicast. For Classic protocol, server support is provided a request/setting message if the server bit is set in the message header. The protocol also includes in its header a destination node field, which should be either the local node#, the related Acnet node#, a multicast form of `09Fx`, or zero.

For Acnet protocols, a server mode is supported without requiring that the requesting node supply a server bit as part of the protocol. The request is "previewed" before processing begins in earnest, so it can be determined how the request should be supported—whether server support is required. As part of this determination, we must know whether the request was multicast or unicast. If it was unicast, and if any of the device data requested is sourced from other front ends, according to analysis of the SSDN, then server support is provided for the request; i.e., the request message is forwarded to a multicast target address so that the rest of the nodes in the project can have a look at it, after which each such node that finds any of its own device data included in the request gears up to return a reply of only its own data. If the original request was unicast, but all device data is sourced from a single other node, then the forwarding is unicast to that single node. If a request is received that includes only the local node's data, the request is initialized in the usual way to return only local data, independent of how the request was received. To determine whether a message was received via multicast or unicast, the destination node field in the Acnet header is checked. If it is of the form `xxFx`, then it was multicast, else it was unicast. (Under IP, the form of this multicast node number is `09Fx`.)

In lieu of being able to determine to what destination physical address a received message was delivered, we can decide the matter in a different way, by examining the destination node field in the Acnet header. If it is of the form `09Fx`, it means that it was multicast. This may only be true for IRMs, but that should be enough, since Vax consoles do not support IP multicasting, and only IRM servers send such multicast request messages when they forward requests.

Booting after power on

On occasions when the lab electricity is out, all IRMs cease operation. When

power is restored, all IRMs attempt to perform a TFTP protocol boot via the network. But the network itself, with its router and switch hardware, may take a long time before it comes up, even many minutes or even hours. Current IRMs have no problem with this; when the network comes up enough so that they can reach the boot server node, within two minutes they will succeed in booting. (This is because a failure by the 162Bug TFTP client results a retry two minutes later.)

VxWorks boot support operates in a different way. If a boot attempt fails, it just quits trying. This means that following a lab site-wide loss of power, and subsequent recovery of same, every front end using network booting will fail, requiring a visit to each and every node to manually reset it. Some front ends, knowing this, may devise elaborate schemes to operate the reset button remotely. We are currently pursuing an effort to rectify this situation, so that network booting under VxWorks will retry until the network is able to support the boot operation.