VxWorks ARP Logic

Analysis effort

Thu, Jun 28, 2001

Introduction

Much of the network behavior details in VxWorks is mysterious. Without access to the source code, we can only make measurements to determine its characteristics. In the PowerPC version of the front-end system code, we make use of the `etherAddrResolve()` function to find out whether a given IP address is present in the VxWorks arp cache. If it is not, we know that `etherAddrResolve()` will already have initiated an ARP request message, hoping that it can fill the arp cache soon with the physical address returned in the ARP reply. Based upon whether we learn that the IP address was already present in the arp cache, we pass the datagram to one of two higher-priority transmitter tasks, each of which can call the `sendto()` function. One task is used for transmitting datagrams when the arp cache already contains the IP address in question; the other is used when the arp cache has yet to receive the ARP reply. The reason for performing this action was that we did not want to inhibit transmissions to other nodes while awaiting an ARP reply from one node. We hoped that either transmitter task would block while awaiting completion of the `sendto()` call, so that the rest of the system code would be free to build more datagrams to be transmitted.

Through analysis of task execution, we have learned that `sendto()` does not block in the case that the arp cache already contains the target physical address. We don't yet know whether it blocks when the arp cache does not contain the target address. We don't know because the "delayed transmitter" task calls `etherAddrResolve()` specifying a minimal delay before it calls `sendto()`. The delays are used to wait for the ARP reply and are specified in VxWorks ticks of 60 Hz. A wait of one tick can result in a very short delay, or it can be a delay up to 16 ms. It does not appear that the arrival of an ARP reply message hastens the return after the delay. So we are forced to wait for the delay times even if the ARP reply arrives very quickly.

This is important because the front-end system presumes to operate in a reliable 15 Hz environment, including network communications. The network hardware can do this already, but we must also be concerned about network software. This is why we are focusing attention on analyzing VxWorks ARP behavior.

Former system

The former front-end system software dealt with ARP delays in a different way. Messages destined for a node for which the physical address was not known were queued while an ARP reply was pending. Upon arrival of the ARP reply, any queued messages were quickly dequeued and dispatched to the target node. The delays due to performing ARP requests was thus minimized, and its impact on communication with other nodes was minimized.

Cause for concern

We observe a number of errors logged by the Acnet system for all of the upgraded PowerPC/VxWorks front-ends in Linac. Do these have anything to do with ARP-related delays? We must make measurements to determine this.

Measurement plan

Adding diagnostics to the delayed transmitter task would be a good way to get a handle on what delays occur. That task would not usually be used, as it is only used when a node is being targeted for which there is no corresponding entry in the VxWorks arp cache. We know that we typically get pairs of such task executions, with the first one occurring as the datagram

reference is written into a message queue on which the task waits. The second one occurs after the delay caused by the `etherAddrResolve()` call. It would be useful to know how much time elapsed between the first and second executions.

There is be a variation on this approach that can be done without modifying the delayed transmitter task code. It takes advantage of the task timing diagnostics built-in to the system code. If a data stream called TASKLOG is defined in a system, the activity of all tasks are logged via entries in that data stream. Using 4K bytes for the queue, we may have only the last second of task activity available at once, but it may be enough for specific tests.

Consider a local application called TCAP that captures task timing diagnostics from the data stream queue at carefully-selected times. This LA could monitor (at 15 Hz) the task counter mantained in the system task table in low memory at 0xe00700. If this counter changes, the delayed transmitter task must have run. Immediately send a one-shot request for a reasonable number of the latest entries in the TASKLOG data stream queue. Scan these entries for those that pertain to the delayed transmitter task. Log the results for later analysis.

Parameters for this LA can include the target node and memory address of a location in memory to be monitored, a memory address for housing the captured result data, the maximum number of such entries in the circular buffer, and the mask for matching the task whose activity is being monitored.

Alternatively, as a first attempt, the parameters might be simply include the task number. The results can be housed in memory local to the LA. The Print Memory page application can be used to list out the results for study.

State logic might include one state that watches for changes in the contents of the task execution counter and issue a one-shot request for task timing diagnostics. Another can await the results of the one-shot request and scan for matches, recording each one found.

In case two one-shots occur close to each other in time, it may be good to avoid logging duplicate entries. If the times match, we can assume that they are the same entry.

After implementation

The parameters used for this "experiment" are the following:
Enable Bit
Target node
Task# in range 0–14 (delayed transmitter is task 14)
Data stream index# (the TASKLOG data stream is normally index 5)
Memory address from which 4 bytes are also logged with the monitored word

The results array are found in the TCAP local application static memory at offset 0x040. The array is sized for 128 entries, each of 16 bytes with the following format:

ipAddress copied from meory address parameter location
task execution counter value that is monitored for changes
task execution time in microsec
time in usual BCD format, but with the last byte in half ms units (in binary)

Use the Print Memory page application to print from address specified as `TCAP+040`.

The program operates in two states. In state 0, it monitors at 15 Hz a data word copied from the specified task execution counter. It also collects the present value of the memroy pointed to by

the specified memory address. When it detects a change, it issues a one-shot request to the target node for the last 120 task timing records. Then it sets state 1.

On the next cycle, in state 1, the one-shot reply data is captured and scanned for a match with the task mask formed from the specified task number. (This is 0x4000 for task #14.) For each entry that matches, the record is copied into the results circular buffer. The first three words are overwritten with the captured memory contents above, plus the value of the monitored word that is the task execution counter. A small attempt is made to avoid duplicating result entries by checking the previous one for a match on the last 4 bytes of the time. If they are the same, it is presumed to be a duplicate, so it is not copied into the results array. Here is an example of some results from monitoring activity in node0611:

```
FILE<0509>  TCAP+040   06/28/01 1503
 0509:000380DE   83E1 7998 403E 0022 0106 2814 4405 1118
     :000380EE   83E1 7998 403E 003F 0106 2814 4405 1136
     :000380FE   83E1 7998 403F 0022 0106 2814 4405 1118
     :0003810E   83E1 7998 403F 003F 0106 2814 4405 1136
     :0003811E   83E1 832F 4041 0022 0106 2814 5002 0141
     :0003812E   83E1 832F 4041 0040 0106 2814 5002 0143
     :0003813E   83E1 8310 4043 0021 0106 2814 5419 0419
     :0003814E   83E1 8310 4043 0040 0106 2814 5419 041D
     :0003815E   83E1 832E 4045 0023 0106 2814 5732 093C
     :0003816E   83E1 832E 4045 0040 0106 2814 5732 0946
     :0003817E   83E1 1052 4047 0021 0106 2814 5936 0580
     :0003818E   83E1 1052 4047 0048 0106 2814 5936 0602
     :0003819E   83E1 7978 4048 001F 0106 2815 0012 0815
     :000381AE   83E1 7978 4048 003B 0106 2815 0012 081E
     :000381BE   83E1 8850 404B 001F 0106 2815 0034 0618
     :000381CE   83E1 8850 404B 0039 0106 2815 0034 061A
```

The first pair of entries seems to exhibit duplication, so that part of the logic isn't perfect. the list of such data may give us an idea of how often the arp cache entries are flushed, and whether they get flushed periodically or after a timeout of no transmissions.

Look at the last bytes of each entry in the results log. Subtract these values in a pair of entries to get the elapsed time between the two delayed transmitter task executions, which should imply the delay time that was used while hoping to receive an ARP reply. These final bytes are in half ms units.

The third word in each results record is the monitored task execution counter. It usually moves in steps of 2, reflecting the execution behavior of that task, as described earlier.

Note that the task execution times are often limited to about 60–70 us. This is apparently the time of the `sendto()` call.