

CSIS0234B Computer and Communication Networks (Class B)

Reading for Tutorial 5

Peeping at the network

In many cases, you have a network program or system which claims that it performs some particular functions, but you wonder how it is done. Perhaps you want this information to help you create your own implementation. Or perhaps you just have a problem that the program doesn't work as expected, and you want to see what is the problem. One would then want to capture frames sent by the program.

1. Who do the capturing

The operating system is responsible for processing all data frames received by network cards and from serial ports, and processes them to see whether it should be forwarded to the network layer, forwarded to another link, processed internally, or simply discarded. Hence the operating system sees and processes every single data frame that is received by any networking device. Due to the needs of capturing network frames, most operating systems have programming interfaces for copying frames that it received to a user program, so that the frames can later be analysed. However, there is no standard here: every OS uses a different programming interface.

As a note, for Ethernet, the physical link is a broadcast media, so it is even possible to trap everything in the network. Normally, the network card will try to be smart: if a frame does not seem to be destined to the computer (because the frame has a destination address which is neither the address of the network card, nor a broadcast or multicast address), the frame is not sent to the computer, so the operating system will not see them. This reduces the amount of frames that must be processed by the operating system, thus giving more CPU cycles to applications. However, if an administrator wants to see the traffic of the network, the Ethernet card can be set into a "promiscuous mode". In this mode, the checking we just mentioned is not performed, so the operating system will receive every frame on the network. At times this is a good way to debug network problems.

2. Selecting frames to capture

It is usually impractical to capture and display all traffic that is going on in a physical link: there are simply too many frames. All tools that capture frames need facilities for filtering unneeded frames. One of these tools, `tcpdump`, has the filtering and capturing code written as a library (`libpcap`, "packet capture library"). As is mentioned above, different operating systems use different programming interface for capturing frames. The library has code to deal with each such interface. This library, and hence the syntax for filtering frames, is reused by many other programs that captures network frames. So although `tcpdump` is completely text-based, and might not be the tool of choice for many administrators (we will use a GUI version `ethereal`), most do learn the syntax that it provides for filtering frames.

The `tcpdump` program is usually invoked by `tcpdump expr`, where *expr* is an expression specifying the capture filter. (If the filter is missing, every frame will be captured and displayed.) The expression is evaluated for each frame, and if it is "true", the frame is selected and displayed. Otherwise the frame is silently dropped. There is an unlucky fact, though: the man page is a rather poor and confusing attempt at explaining the syntax of the expression. So instead of forcing you to read the man page, we highlight the main features here.

An expression is a string consisting of one or more **primitives** combined using logical operators like `and` (or `&&`), `or` (or `||`) and `not` (or `!`). There are only two levels of precedence: `not` is higher than `and` and `or`. Everything else is left-to-right. If you don't like the result, you can use parentheses to enclose them. Those using `tcpdump` usually put the whole expression in a pair of single quotes to prevent the shell program to interpret them. E.g.,

```
tcpdump 'ip && !(host 127.0.0.1)'
```

contains two primitives `ip` and `host 127.0.0.1`, combined in such a way that the frame is selected if the first is true and the second is false. It remains to explain what is a primitive.

All primitives work by checking whether the frame satisfy some criteria. The basic primitive is a **relational** operation: find some particular bytes of the frame, perform some operations to form an **arithmetic expression**, and match it against another arithmetic expression using a relational operator like `==`, `!=`, `>`, etc. Arithmetic expressions are formed by things that you would expect: composing values using `+`, `-`, `*`, `/`, bitwise `&`, and bitwise `|`; overriding precedence by parentheses. Values can be a constant, or the word `len` denoting the length of the frame, or the content of the frame. The frame content is denoted by a special syntax `proto[start:size]`, saying that you want to get the `size` byte integers starting from the `start`-th byte of the protocol layer `proto` of the frame. The `size` part may be omitted, in which case it means 1 byte. E.g., the following returns true if the frame is sent to an IP multicast address, i.e., with first byte of the IP destination field (16-th byte in the IP header) at least 224, but the whole IP destination is not 255.255.255.255 (i.e., 0xffffffff):

```
tcpdump 'ip[16] >= 224 and ip[16:4] != 0xffffffff'
```

This is clumsy, and requires you to know the exact location of the required field of the frame. There are various **shorthands** for writing them in a more readable way.

3. Various shorthands

There are many shorthands, and we will explain the most important ones. Most shorthands has a common form where some **keywords** are followed by an **id** like `www.csis.hku.hk` or `53`. The earlier example `host 127.0.0.1` is of this form: the `host` keyword is followed by the id `127.0.0.1`. If a keyword is missing, then it is assumed to be the set of keywords in the last shorthand. E.g., `host 127.0.0.1` or `172.16.0.1` means `host 127.0.0.1` or `host 172.16.0.1`. Some of the possible keywords are:

- `host hostname`: true if the IP header has `hostname` in either the source or destination address. One can be more specific about the direction by preceding the expression by `dst`, `src`, `dst or src` or `src or dst` and `src`. One can also add an Ethernet protocol name (one of `ip`, `arp` and `ip6`) to specify what kind of network protocol you are interested in. So

```
arp src host 0.0.0.0
```

will get all ARP requests (see the last section) with source host 0.0.0.0. The `hostname` can also be specified as a DNS name, which will be searched through the normal process using `getaddrinfo()`.

- `port portnum`: true if the IP header has `portnum` as the source or destination port number. Again, `dst`, `src`, `dst or src` or `src or dst` and `src` can be used to be specific about the source and destination. One can also insert an IP protocol name (one of `tcp` and `udp`) to specify

what kind of protocol you are interested in. So

```
tcp port 53
```

will get all the DNS (Domain Name Server, port 53) traffic that are done via TCP rather than the normal UDP. The *portnum* (53) can also be specified by a name found in `/etc/services` (in this case, `domain`).

- `proto protonum`: true if the frame is for IP protocol *protonum*. E.g., `proto 17` will trap all frames containing UDP packets. Similar to the case for port numbers, *protonum* can also be specified by a name found in `/etc/protocols`, so you can write `proto UDP` instead. Most likely you won't even do that: there is an abbreviation `udp` which stands for it. The other such abbreviations include `tcp` and `icmp`. Note that `tcp`, `udp`, `icmp`, etc. are also keywords, they need to be escaped via a backslash; e.g., `proto \udp`.

The whole expression can be preceded by the keyword `ether`, in which case it chooses frames based on Ethernet protocol number. The only ones important for us are `0x800` and `0x806` (IP and ARP). You don't really need to type these numbers, as you can use the abbreviations `ip` and `arp` instead of the more verbose `ether proto 0x800` and `ether proto 0x806`.

- `broadcast` or `multicast`: true for Ethernet broadcast or multicast frames. This can be preceded by the keyword `ip` (i.e., `ip broadcast` or `ip multicast`) in case you want only broadcasts or multicasts for IP frames, not ARP or other (non-IP) protocols.

With all these knowledge, we know the first example `ip and not host 127.0.0.1` means an IP frame that has neither source nor destination address being the address `127.0.0.1`, which is the loopback address.

There are other primitives that are not as useful when peeping at an IP network, and we choose to let you view the man page when the need comes. There is one extra keyword, `net`, that is also important, but we decided not to explain it here as it won't make sense until you know the Internet addressing scheme.

4. A packet capturing exercise: ARP

As an exercise, we will capture ARP frames in the lab during the tutorial. To prepare you for it, this is a really short introduction to the protocol.

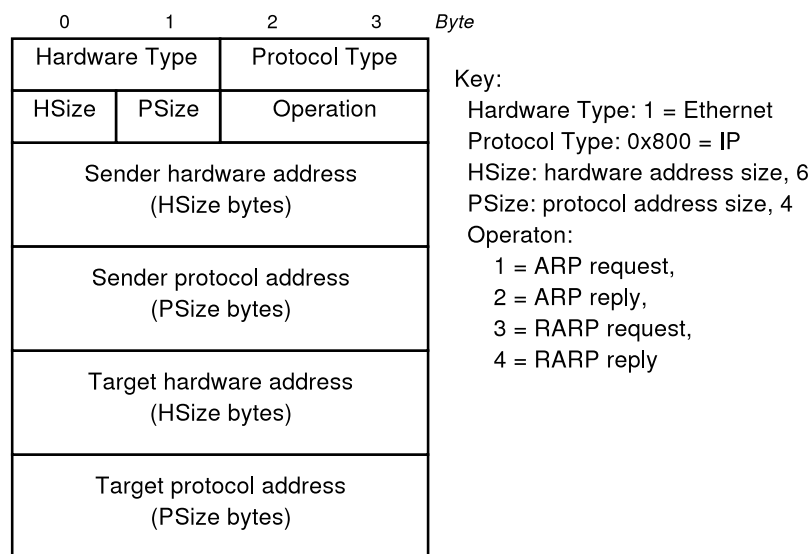
ARP, the Address Resolution Protocol, is used when a host wants to send an IP packet to another host in the same network. So the source computer builds a packet with the IP address of the destination packet, the IP address either specified by the user, or is obtained from the DNS server (during `gethostinfo()`). Once the IP packet is ready, the OS passes the packet down to layer 2 (datalink layer), asking it to send a frame. But... to whom? For point-to-point links this is not a problem (there is only one receiver), but for broadcast links like the Ethernet, we have to be more specific.

Of course, to the one holding the 32-bit destination IP address! But the datalink hardware (i.e., Ethernet card) knows nothing about the IP packet structure. It is a design decision: the lower datalink layer should not be dependent upon the upper layer, so that if you decide to switch from IP (Internet) to IPX (Novell network) or AppleTalk (Mac), your network card will still serve you perfectly.

So, how about just broadcast the frame to everybody? In every computer in the network, the OS checks whether the destination IP address of the packet refers to itself. If not, the packet would be dropped. This works, and indeed sometimes the network operate this way. But it is not very efficient, since every frame must be processed by the CPU of every computer.

ARP fills this gap. ARP is a protocol on top of the link-layer protocol (i.e., Ethernet protocol). Essentially, the computer which wants to send an IP packet would broadcast an **ARP request**, asking essentially **who owns this IP?** The destination computer who owns the IP address respond with an **ARP reply** that contains its 48-bit hardware address. Al other computers drop the ARP query. Once the sender knows the target address, the network packet can be sent to the correct Ethernet address, so the filtering is done by the Ethernet card rather than the host OS (freeing the CPU for applications, as we have seen at the beginning of the notes).

An ARP packet has the following format:



The RARP operations are not usually used: its functionalities is replaced by DHCP.

Since the Ethernet card knows nothing about IP addresses, the ARP processing must be done by the OS. So instead of asking the CPU of every computer to handle a broadcast IP packet, now we ask the CPU of every computer to handle a broadcast ARP query, and in doing so we increase the number of frames to be sent from 1 to 3, and increase the round-trip-delay likewise. What a good deal! Thus it is not desirable to make one ARP requests for each packet. Instead, each computer should keep an **ARP cache**, containing the replies of each request. One can inspect the ARP cache using the command `arp` command. The computer can work on the cache at various times. The ARP algorithm can also be used when you start the network. We will investigate these issues during the tutorial.