Lecture 11: Packet forwarding

Anirudh Sivaraman

2020/10/19

This week we'll talk about the data plane. Recall that the routing layer broadly consists of two parts: (1) the control plane that computes routes using distributed protocols such as the distance vector and link state protocols, and (2) the data plane that forwards packets based on routes computed by these protocols. The control plane kicks in every time the topology changes (e.g., a new router boots up, a router goes down, or a link cost is modified), while the data plane operates on every packet to look up the packet's destination address and determine its output port.

The interface between the control and data planes is the forwarding table, which is a map/dictionary/hash-table that maps destination IP addresses to the output port connecting to the next hop on the way to that destination. The control plane writes into this forwarding table, while the data plane reads from it. Because the data plane operates on a per-packet basis, it typically needs to be much faster than the control plane. On a high-speed 1 Tbit/s router today, assuming a 1000 bit packet, a packet arrives every 1 ns. It is quite hard to build a lookup table supporting 1 B lookups per second using an off-the-shelf CPU today. Hence, the data plane is built out of dedicated chips (called switching chips), which are essentially extremely large digital circuits within a single package that have been specialized to packet forwarding.

The topic of this lecture will be packet forwarding: how to perform lookups at these high speeds when a new packet arrives every nanosecond. Then, in the next lecture, we'll deal with packet buffering and scheduling, which is the task of storing packets and selecting packets for transmission when more than one input port simultaneously wants to send a packet to the same output port.

OK. Now for packet forwarding. You're given an IP address corresponding to the destination this packet must get to. The routing protocol has already computed a forwarding table that contains this destination. How do you lookup this IP address in the forwarding table to determine the port number of the next hop on the way to this IP address? And how do you do it quickly?

We'll consider two variants of this problem. The first is *exact matching*: a destination IP address is matched against every IP address in the forwarding table to find a unique next hop port—assuming there are no duplicates in the forwarding table. While this works, and is in fact how switches at the MAC layer operate, it suffers from a serious problem. It requires each router to store the next hop for every public IP address in its routing table. With 32-bit IP addressess, this requires every router to store around 4 billion forwarding table entries, which would take up several GBytes of router memory. Memories that are cheap enough to store several GBytes (e.g., DRAM memory found in your desktops) are also typically too slow to support 1 billion lookups per second. Besides, this requires every router to know of every other router/end-host as part of its routing protocol, which is a serious problem.

We have looked at the solution to this in the lecture on inter-domain routing: hierarchy. Hierarchy allows us to separate inter and intradomain routing by allowing each inter-domain router to only know about other inter-domain routers and each intra-domain router to only know about other routers within its own domain. This separation reduces (1) the volume of traffic generated by the routing protocol because each routing protocol (whether inter or intradomain) involves fewer entities, and (2) the size of the forwarding tables, again because each router needs to forward packets to fewer other routers.

But, as a result of this separation, a router deep inside the network only knows how to get to the *network* of a particular destination address. It *does not know how to get to the address itself*. These networks are represented using a format similar to IP addresses, but with the allowance that some bits can be wildcards. For instance, 8.8.8.* represents the network with IP addresses between 8.8.8.1 and 8.8.8.255, because the * can stand for any octet between 1 and 255.[1] Now, if a router gets a packet destined for 8.8.8.4, the router needs to determine that it belongs to the network 8.8.8.*, and then forward it out through the next hop for that network.

This process of matching a destination IP address against a set of networks, as opposed to a set of IP addresses, is called *prefix matching* because we are checking whether or not the network and the destination IP address share a common prefix. In the case of the destination IP address 8.8.8.4 and the network 8.8.8.*, the address and the network share the prefix 8.8.8. But once we move to prefix matching from exact matching, there is no longer guaranteed to be one unique network for an incoming destination IP address. For instance, the networks 8.8.*.* and 8.8.8.* both share a prefix with the destination address 8.8.8.8 (8.8 and 8.8.8 respectively) even though the networks themselves are different. The Internet Protocol resolves this by enforcing that the longest prefix is the one that should be chosen to determine the next hop because it is likely to be closer to the destination address.

Before we discuss the core algorithms behind both exact and longest-prefix matching, it is useful to settle on some terminology.

1 Hardware vs. software implementations

When we say that we are implementing a piece of functionality in software, it means we are writing an algorithm for that functionality; this algorithm is intended to be run on an off-the-shelf processor such as the Intel Core i7. When we say that we are implementing a piece of functionality in hardware, it means we are building a custom digital circuit for that functionality out of boolean gates—even though our description of hardware will be at a considerably higher level of abstraction than gates. As a simple example, the programs you implement for this class are implemented in software, but the video decoding routines on your mobile phones run on dedicated hardware devices on your phone called video coprocessors or video accelerators. As another example, a digital circuit for multiplying 2 32-bit integers (which is at the heart of Intel's MUL assembly instruction) is an example of doing things in hardware. On the other hand, if you implement multiplication by writing an algorithm/program in Python to repeatedly add a number to itself a certain number of times, then you are doing things in software.

To a first approximation, software implementations offer flexibility (you can write a Python program to do anything you wish), but come at the cost of determinism and high performance. Hardware implementations are rigid because there are only a certain set of operations that can be done in hardware at a certain clock frequency. In return, they provide determinism and high performance. You can follow the rest of this lecture even if you don't understand any hardware so long as you remember that a hardware implementation is typically characterized by deterministic guarantees on performance (e.g., deterministic look up times in the case of longest-prefix or exact matching), while a software implementation typically involves clever algorithms that provide good average-case guarantees at the cost of determinism.

2 Exact matching

The task of exact matching is quite similar to the functionality of a hash table that you have probably encountered in your basic algorithms course. One way to build a hash table is to use an array with a certain number of elements/slots/buckets, where each slot can contain a number of keys and their associated values. In a typical implementation of a hash table with *chaining*, each incoming element is passed through a hash function that maps the element to one of the array slots. A good hash function will ensure that elements are uniformly distributed across the array slots.

During an insert, the incoming element is appended to the end of a list at the particular array slot that the incoming element hashes to, i.e., it is chained together with the existing elements at that slot. During a lookup, an incoming element is again first hashed to a particular slot. The element's key is then sequentially compared with the keys of all elements stored within the list at that particular slot. If the element matches against a particular key, the value at that list index is returned as the value for the lookup operation.

[1]Each wildcard character strictly stands for a particular bit, but we'll let it stand for a whole octet to simplify our notation.

The problem with this approach is that traversing a list during lookup is a non-deterministic operation that is antithetic to high-speed hardware implementations. One really obvious way to fix this non-determinism is to cap the size of the list at each slot so that the traversal time is bounded. But then, what happens when a new element needs to be inserted into a full slot? And how likely is it that an element hashes to a full slot if elements are assigned to hash slots at random by the hash function?

With the standard hash table as presented above, it turns out that the size of the largest slot grows relatively quickly with the number of elements. In other words, it is quite likely that the largest slot will hit the cap that we have set on the size of the list at each slot. It turns out that a better solution that slows down the growth rate of the largest slot is to hash to two slots using two independent hash functions, and then insert the element into whichever slot has fewer elements. This idea of picking *two* locations at random and picking the one that is smaller recurs in many other contexts in computer systems and is called the power of two choices [4]. The fascinating aspect of this result is that picking the smallest of two slots gives us a big improvement in terms of decreasing the size of the largest slot for a given number of elements, but, going from the smallest of two to smallest of three, four, or five provides us diminishing returns. The power of two choices applied to hashing is called 2-choice hashing [1]. When there are $d > 2$ choices, we call it d-choice hashing.

A slight twist on 2-choice hashing provides us an *even* better design that slows down the rate of growth of the largest slot even further. This design is called *2-left* hashing [2] where a single hash tables is divided into 2 independent sub hash tables with 2 independent hash functions. When an incoming element is to be inserted into the hash table, it is hashed into 2 indices for the 2 sub tables and placed into the index that has the minimum number of elements. If two indices are tied for the minimum number of elements, we always break ties towards the left, i.e., if subtables 1 and 2 were tied, we would always pick 1 [5]. 2-left hashing can be generalized to d-left hashing where we use d subtables with d independent hash functions. Typical implementations of exact matching today use d-left hash tables with d=2 or d=4.

It is important to note that whether we use (1) a standard hash table with a cap on the number of elements in each slot, or (2) d-choice hashing, or (3) d-left hashing, we have to confront the situation where eventually an element hashes into a slot that is already full. d-left hashing delays the occurence of this phenomenon relative to a standard hash table, but still needs to deal with this eventuality. The general approach to this problem is alert the control plane that the forwarding table could not insert this new entry and let the control plane deal with this problem.

3 Longest-prefix matching

Next, we'll look at the problem of longest-prefix matching. This problem is similar to the autocomplete problem that search engines like Google deal with. Here, a user's search query is compared against a stored database of candidate queries to determine the candidate query with the longest shared prefix with the user's search query. The candidate query(s) with the longest shared prefix are displayed as autocomplete suggestions.

A common software implementation of longest-prefix matching is a data structure called a trie (Figure 1) [3] that stores strings (in our case IP addresses) in a tree structure where: (1) the root of the tree is associated with an empty string and (2) all descendants of a tree node share a common prefix; this prefix is the string stored at the tree node. Then an incoming IP address can be compared against the IP addresses of all networks stored in this trie until it finds a network with the longest prefix match. The problem with this approach, again, is the lack of determinism. The lookup time depends on how long the longest prefix for a particular destination is.

Instead, to guarantee determinism, we use a different approach. We use a hardware mechanism that is known as a content addressable memory (or CAM for short). CAMs are a particular kind of memory where the memory is supplied a piece of content, as opposed to an address, and the value associated with that memory is returned. You can think of a CAM as the hardware equivalent of a map/hash table/dictionary. To provide constant and deterministic lookup times that are independent of the number of entries in the CAM, a CAM compares an incoming piece of content against all CAM entries in parallel to find a match. This parallel comparison increases the power consumption of a CAM but provides deterministic lookup times.

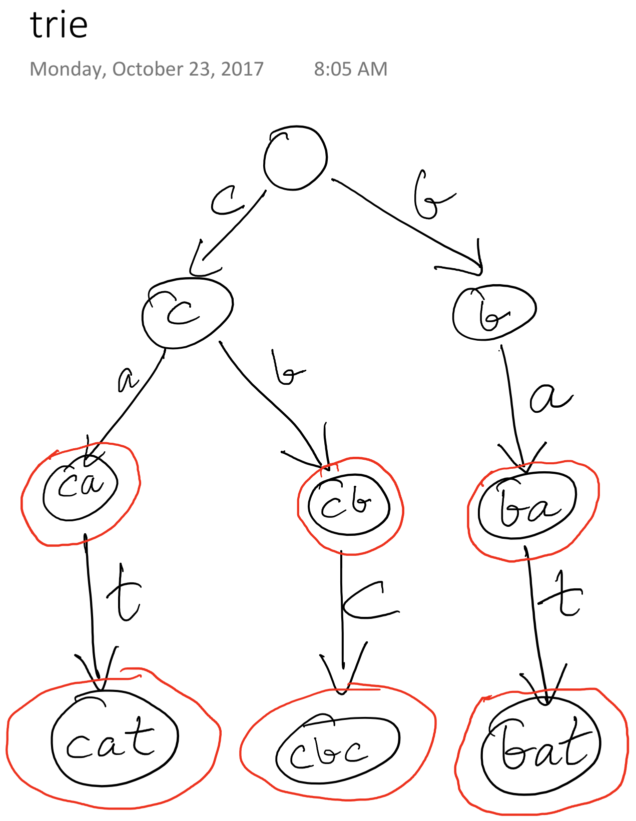

Figure 1: Trie with six entries: ca, cb, ba, cat, cbc, and bat. Entries are circled in red.

A CAM (also called a binary CAM) can be used for exact match lookups.[2] But for prefix matching, we need a different kind of CAM called a ternary CAM (TCAM) that allows some TCAM entries to be wildcard (or don't-care) bits. To illustrate this, let's consider a 4-bit IP address of the form b1.b2.b3.b4, where b1 through b4 is either 0 or 1. To match an incoming IP address, 1.0.0.1, against a network 1.0.0.*, we add an entry to the TCAM with the content 1, 0, 0, and * at each of the 4 bits. So instead of checking for equality between every bit of the input and every bit in the TCAM entry, we check for equality only on the non-wildcard bits. With this modification, 1.0.0.1 matches against 1.0.0.*.

To handle longest-prefix matching—as opposed to just prefix matching—we order TCAM entries in increasing order of the number of non-wildcard bits in each entry. By this ordering, we would insert the TCAM entry 1.1.*.* before 1.1.0.* because 1.1.*.* has 2 non-wildcard bits, while 1.1.0.* has 3. Then, we find the largest entry index among all the TCAM entries that match against an incoming IP address.

4 Beyond destination-address-based forwarding

So far our discussion has focused on forwarding based on destination IP address (or destination MAC address in the case of switches). In other words, we lookup the destination IP address in an exact or prefix match lookup table to determine the next hop. We can generalize this to other packet headers beyond the destination IP address. For instance, we may want to route a packet differently based on its source IP address or TCP source/destination port number or some other header in the packet.

We may want to do this for several reasons. We may want to provide deferential treatment to certain packets if the users have paid for such a service. We may also want to isolate traffic from two different users

[2] In practice, a binary CAM is rarely used for exact matches because d-left hash tables consume less logic gate area than CAMs when implemented as a digital circuit.

so that their packets never traverse the same network link for privacy reasons. We might want to drop traffic originating from some source IP addresses or destined to certain destination TCP ports.

The general term for such routing that goes beyond looking at the destination address is policy routing. It is one of the characteristics of software-defined networking (SDN), an approach to making networks more flexible by allowing operators to specify a variety of policy routing schemes using a consistent interface to the forwarding tables on a router called OpenFlow. We'll have more to say about SDN later in the course. For now, though, the important thing is that whether it is policy routing or destination-address-based forwarding, the underlying data plane mechanisms are the same: exact or prefix match lookup based on *some* packet header. The only difference is in what packet headers are used during the match process.

References

[1] 2-choice hashing. `https://xlinux.nist.gov/dads/HTML/twoChoiceHashing.html`.

[2] 2-left hashing. `https://xlinux.nist.gov/dads/HTML/twoLeftHashing.html`.

[3] trie. `https://xlinux.nist.gov/dads/HTML/trie.html`.

[4] Michael David Mitzenmacher. *The Power of Two Choices in Randomized Load Balancing*. PhD thesis, University of California, Berkeley, 1996. AAI9723118.

[5] Berthold Vöcking. How Asymmetry Helps Load Balancing. *Journal of the ACM*, 2003.