CHAPTER

Exact-match lookups

10

"Challenge-and-response" is a formula describing the free play of forces that provokes new departures in individual and social life. An effective challenge stimulates men to creative action.

—Arnold Toynbee

In Part 3, for simplicity of terminology, we will generically refer to interconnect devices as *routers*. Each chapter in Part 3 addresses the efficient implementation of a key function for such routers. In the simplest model of a router forwarding path the destination address of a packet is first looked up to determine a destination port; the packet is then switched to the destination port; finally, the packet is scheduled at the destination port to provide QoS (Quality of Service) guarantees. In addition, modern high-performance routers also subject packets to internal striping (to gain throughput) and to internal credit-based flow control (to prevent loss on chip-to-chip links). The chapters are arranged to follow the same order, from lookups to switching to QoS.

Thus the first three chapters concentrate on the surprisingly difficult problem of state lookup in routers. The story begins with the simplest exact match lookups in this chapter, progresses to longest-prefix lookups in Chapter 11, and culminates with the most complex classification lookups in Chapter 12.

What is an exact-match lookup? Exact-match lookups represent the simplest form of database query. Assume a database with a set of tuples; each tuple consists of a unique fixed-length key together with some state information. A query specifies a key K. The goal is to return the state information associated with the tuple whose key is K.

Now, exact-match queries are easily implemented using well-studied techniques, such as binary search and hash tables (Cormen et al., 1990). However, they are still worth studying in this book for two reasons. First, in the networking context the models and metrics for lookups are different from the usual algorithmic setting. Such differences include the fact that lookups must complete in the time to receive a packet, the use of memory references rather than processing as a measure of speed, and the potential use of hardware speedups. Exact-match lookups offer the simplest opportunity to explore these differences. A second reason to study exact-match lookups is that they are crucial for an important networking function, called *bridging*,[1] that is often integrated within a router.

We are grateful to Michael Mitzenmacher for proofreading Section 10.3.3 that describes the d-left scheme.

This chapter is organized around a description of the *history* of bridges. This is done for one chapter in the book, in the hope of introducing the reader to the *process* of algorithmics at work in a real

[1] A device commonly known as a LAN switch typically implements bridge functionality.

Network Algorithmics. https://doi.org/10.1016/B978-0-12-809927-8.00017-8

Table 10.1 Principles used in the various exact-match lookup techniques discussed in this chapter.

Number	Principle	Used in
P15	Use efficient data structures: binary search table	First bridge
P5	Hardware FPGA for lookup only	
P15	Use efficient data structure: perfect hashing	Gigaswitch
P2a	Precompute hash function with bounded collisions	FDDI bridge
P5	Pipeline binary search	

product that changed the face of networking. This chapter also describes some of the stimuli that lead to innovation and introduces some of the people responsible for it.

Arnold Toynbee (Toynbee and Caplan, 1972) describes history using a challenge–response theory, in which civilizations either grow or fail in response to a series of challenges. Similarly, the history of bridges can be described as a series of three challenges, which are described in the three sections of this chapter: Ethernets Under Fire (Section 10.1), Wire Speed Forwarding (Section 10.2), and Scaling Lookups to Higher Speeds (Section 10.3). The responses to these challenges led to what is now known as 802.1 spanning tree bridges (IEEE Media, 1997).

The techniques described in this chapter (and the corresponding principles) are summarized in Table 10.1.

Quick reference guide

The implementor interested in fast exact-match schemes should consider either parallel hashing techniques inspired by perfect hashing (Section 10.3.1) or d-left (Section 10.3.3), or pipelined binary search (Section 10.3.2).

10.1 Challenge 1: Ethernet under fire

The first challenge arose in the late 1980s. Ethernet, invented in the 1970s as a low-cost, high-bandwidth interconnect for personal computers, was attacked as behaving poorly at large loads and being incapable of spanning large distances. Recall that if two or more nodes on an Ethernet send data at the same time, a collision occurs on the shared wire. All senders then compute a random retransmission time and retry, where the randomization is chosen to minimize the probability of further collisions.

Theoretical analyses (e.g., Bux and Grillo, 1985) claimed that as the utilization of an Ethernet grew, the effective throughput of the Ethernet dropped to zero because the entire bandwidth was wasted on retransmissions. A second charge against Ethernet was its small distance limit of 1.5 km, much smaller than the limits imposed by, say, the IBM token ring.

While the limited-bandwidth charge turned out to be false in practice (Boggs et al., 1988), it remained a potent marketing bullet for a long time. The limited-distance charge was, and remains, a true limitation of a single Ethernet. In this embattled position network marketing people at Digital Equipment Corporation (DEC) around 1980 pleaded with their technical experts for a technical riposte to

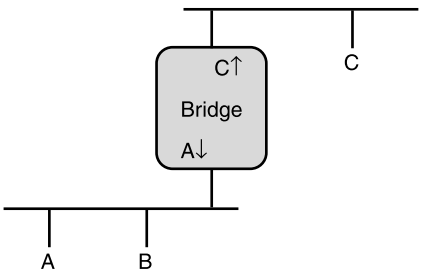

FIGURE 10.1

Toward designing a bridge connecting two Ethernets.

these attacks. Could not their bright engineers find a clever way to "extend" a single Ethernet such that it could become a longer Ethernet with a larger effective bandwidth?

First, it was necessary to discard some unworkable alternatives. Physical layer bit repeaters were unworkable because they did not avoid the distance and bandwidth limits of ordinary Ethernets. Extending an Ethernet using a router did, in theory, solve both problems but introduced two other problems. First, in those days routers were extremely slow and could hardly keep up with the speed of the Ethernet.

Second, there were at least six different routing protocols in use at that time, including IBM's SNA, Xerox's SNS, DECNET, and AppleTalk. Hard as it may be to believe now, the Internet protocols were then only a small player in the marketplace. Thus a router would have to be a complex beast capable of routing multiple protocols (as Cisco would do a few years later), or one would have to incur the extra cost of placing multiple routers, one for each protocol. Thus the router solution was considered a nonstarter.

Routers interconnect links using information in the routing header, while repeaters interconnect links based on physical-layer information, such as bits. However, in classical network layering there is an intermediate layer called the data link layer. For an Ethernet, the data link layer is quite simple and contains a 48-bit unique Ethernet destination address.[2] Why is it not possible, the DEC group argued, to consider a new form of interconnection based only on the data link layer? They christened this new beast a data link layer relay, or a *bridge*.

Let us take an imaginary journey into the mind of Mark Kempf, an engineer in the Advanced Development Group at DEC, who invented bridges in Tewksbury, MA, around 1980. Undoubtedly, he drew something like Fig. 10.1, which shows two Ethernets connected by a bridge; the lower Ethernet line contains stations *A* and *B*, while the upper Ethernet contains station *C*.

The bridge should make the two Ethernets look like one big Ethernet so that when *A* sends an Ethernet packet to *C* it magically gets to *C* without *A*'s having to even know there is a bridge in the middle. Perhaps Mark reasoned as follows in his path to a final solution.

Packet Repeater: Suppose *A* sends a packet to *C* (on the lower Ethernet) with destination address *C* and source address *A*. Assume the bridge picks up the entire packet, buffers it, and waits for a

[2] Note that Ethernet 48-bit addresses have no relation to 32-bit Internet addresses.

transmission opportunity to send it on the upper Ethernet. This avoids the physical coupling between the collision-resolution processes on the two Ethernets that would be caused by using a bit repeater. Thus the distance span increases to 3 km, but the effective bandwidth is still that of one Ethernet because every frame is sent on both Ethernets.

Filtering Repeater: The frame repeater idea in Fig. 10.1 causes needless waste (**P1**) when A sends a packet to B by sending the packet unnecessarily on the upper Ethernet. This waste can be avoided if the bridge has a table that maps station addresses to Ethernets. For example, suppose the bridge in Fig. 10.1 has a table that maps A and B to the lower Ethernet and C to the upper Ethernet. Then on receipt of a packet from A to B on the lower Ethernet, the bridge need not forward the frame because the table indicates that destination B is on the same Ethernet the packet was received on. If, say, a fraction p of traffic on each Ethernet is to destinations on the same Ethernet (locality assumption), then the overall bandwidth of the two Ethernet systems becomes $(1 + p)$ times the bandwidth of a single Ethernet. This follows because the fraction p can be simultaneously sent on both Ethernets, increasing overall bandwidth by this fraction. Hence *both* bandwidth and distance increase. The only difficulty is figuring out how the mapping table is built.

Filtering Repeater With Learning: It is infeasible to have a manager build a mapping table for a large bridged network. Can the table be built automatically? One aspect of Principle **P13** (exploit degrees of freedom) is Polya's (Polya, 1957) problem-solving question: "Have you used all the data?" So far, the bridge has looked only at *destination addresses* to forward the data. Why not also look at *source addresses*? When receiving a frame from A to B, the bridge can look at the source address field to realize that A is on the lower Ethernet. Over time, the bridge will learn the ports through which all active stations can be reached.

Perhaps Mark rushed out after his insight, shouting "Eureka!" But he still had to work out a few more issues. First, because the table is initially empty, bridges must forward a packet, perhaps unnecessarily, when the location of the destination has not been learned. Second, to handle station movement, table entries must be timed out if the source address is not seen for some time period T. Third, the entire idea generalizes to more than two Ethernets connected together without cycles, to bridges with more than two Ethernet attachments, and to links other than Ethernets that carry destination and source addresses. But there was a far more serious challenge that needed to be resolved.

10.2 Challenge 2: wire speed forwarding

When the idea was first proposed, some doubting Thomas at DEC noticed a potential flaw. Suppose in Fig. 10.1 that A sends 1000 packets to B and that A then follows this burst by sending, say, 10 packets to C. The bridge receives the 1000 packets, buffers them, and begins to work on forwarding (actually discarding) them. Suppose the time that the bridge takes to look up its forwarding table is twice as long as the time it takes to receive a packet. Then after a burst of 1000 back-to-back packets arrive, a queue of 500 packets from A to B will remain as a backlog of packets that the bridge has not even examined.

Since the bridge has a finite amount of buffer storage for, say, 500 packets, when the burst from A to C arrives they may be dropped without examination because the bridge has no more buffer storage. This is ironic because the packets from A to B that are in the buffer will be dropped after examination, but the bridge has dropped packets from A to C that need to be forwarded. One can change the numbers used in this example but the bottom line is unchanged: If the bridge takes more time to forward a packet

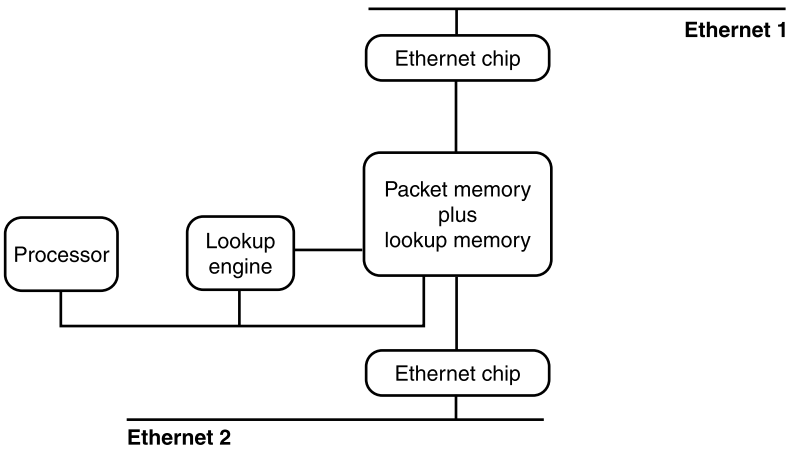

FIGURE 10.2

Implementation of the first Ethernet-to-Ethernet bridge.

than the minimum packet arrival time, there are always scenarios in which packets to be forwarded will be dropped because the buffers are filled with packets that will be discarded.

The critics were quick to point out that routers did not have this problem[3] because routers dealt only with packets addressed to the router. Thus if a router were used, the router–Ethernet interface would not even pick up packets destined for B, avoiding this scenario.

To finesse this issue and avoid interminable arguments, Mark proposed an implementation that would do *wire speed forwarding* between two Ethernets. In other words, the bridge would look up the destination address in the table (for forwarding) and the source address (for learning) in the time it took a minimum-size packet to arrive on an Ethernet. Given a 64-byte minimum packet, this left 51.2 microsecond to forward a packet. Since a two-port bridge could receive a minimum-size packet on each of its Ethernets every 51.2 microsecond, this actually translated into doing two lookups (destination and source) every 25.6 microsecond.

It is hard to appreciate today, when wire speed forwarding has become commonplace, how astonishing this goal was in the early 1980s. This is because in those days one would be fortunate to find an interconnect device (e.g., router, gateway) that worked at kilobit rates, let alone at 10 Mbit/sec. Impossible, many thoughts. To prove them wrong, Mark built a prototype as part of the Advanced Development Group in DEC. A schematic of his prototype, which became the basis for the first bridge, is shown in Fig. 10.2.

The design in Fig. 10.2 consists of a processor (the first bridge used a Motorola 68000), two Ethernet chips (the first bridge used AMD Lance chips), a lookup chip (which is described in more detail later), and a four-ported shared memory. The memory could be read and written by the processor, the Ethernet chips, and the lookup engine.

[3] Oddly enough, even routers have the same problem of distinguishing important packets from less important ones in times of congestion, but this was not taken seriously in the 1980s.

The data flow through the bridge was as follows. Imagine a packet P sent on Ethernet 1. Both Ethernet chips were set in "promiscuous mode," whereby they received all packets. Thus the bits of P are captured by the upper Ethernet chip and stored in the shared memory in a receive queue. The processor eventually reads the header of P, extracts the destination address D, and gives it to the lookup engine.

The lookup engine looks up D in a database also stored in the shared memory and returns the port (upper or lower Ethernet) in around 1.3 microsecond. If the destination is on the upper Ethernet, then the packet buffer pointer is moved to a free queue, effectively discarding the packet; otherwise, the buffer pointer is moved to the transmit queue of the lower Ethernet chip. The processor also provides the source address S in packet P to the lookup engine for learning.

His design paid careful attention to algorithmics in at least three areas to achieve wire speed forwarding at a surprisingly small manufacturing cost of around $1000.

- **Architectural Design:** To minimize the cost, the memory was cheap DRAM with a cycle time of 100 nanosecond that was used for packet buffers, scratch memory, and the lookup database. The four-port memory (including the separate connection from the lookup engine to the memory) and the buses were carefully designed to maximize parallelism and minimize interference. For example, while the lookup engine worked on doing lookups to memory, the processor continued to do useful work. Note that the processor has to examine the receive queues of both Ethernet chips in dovetailed fashion to check for packets to be forwarded from either the top or bottom Ethernets. Careful attention was paid to memory bandwidth, including the use of page mode (Chapter 2).
- **Data Copying:** The Lance chips used DMA (Chapter 5) to place packets in the memory without processor control. When a packet was to be forwarded between the two Ethernets, the processor only flipped a pointer from the receive queue of one Ethernet chip to the transmit queue of the other processor.
- **Control Overhead:** As with most processors, the interrupt overhead of the 68000 was substantial. To minimize this overhead, the processor used polling, staying in a loop after a packet interrupt and servicing as many packets that arrive, in order to reduce context-switching overhead (Chapter 6). When the receive queues are empty, the processor moves on to doing other chores, such as processing control traffic. The first data packet arrival after such an idle period interrupts the processor, but this interrupt overhead is spread over the entire batch of packets that arrive before another idle period begins.
- **Lookups:** Very likely, Mark went through the eight cautionary questions found in Chapter 3. First, to avoid any complaints, he decided to use binary search (**P15**, efficient data structures) for lookup because of its determinism. Second, having a great deal of software experience before he began designing hardware, he wrote some sample 68000 code and determined that software binary search lookup was the bottleneck (**Q2** in Chapter 3) and would exceed his packet processing budget of 25.6 microsecond. Eliminating the destination and source lookup would allow him to achieve wire speed forwarding (**Q3**). Recall that each iteration of binary search reads an address from the database in memory, compares it with the address that must be looked up, and uses this comparison to determine the next address to be read. With added hardware (**P5**), the comparison can be implemented using combinatorial logic (Chapter 2), and so a first-order approximation of lookup time is the number

of DRAM memory accesses. As the first product aimed for a table size of 8000,[4] this required $\log_2 8000$ memory accesses of 100-nanosecond each, yielding a lookup time of 1.3 microsecond. Given that the processor does useful work during the lookup, two lookups for source and destination easily fit within a 25.6 microsecond budget (**Q4**).

To answer **Q5** in Chapter 3 as to whether custom hardware is worthwhile, Mark found that the lookup chip could be cheaply and quickly implemented using a PAL (programmable array logic; see Chapter 2). To answer **Q7**, his initial prototype met wire speed tests constructed using logic analyzers. Finally, **Q8**, which asks about the sensitivity to environment changes, was not relevant to a strictly worst-case design like this.

The 68000 software, written by Bob Shelley, also had to be carefully constructed to maximize parallelism. After the prototype was built, Tony Lauck, then head of DECNET, was worried that bridges would not work correctly if they were placed in cyclic topologies. For example, if two bridges are placed between the same pair of Ethernets, messages sent on one Ethernet will be forwarded at wire speed in the loop between bridges. In response, Radia Perlman, then the DEC routing architect, invented her celebrated *spanning tree* algorithm. The algorithm ensures that bridges compute a loop-free topology by having redundant bridges turn off appropriate bridge ports.

While you can read up on the design of the spanning tree algorithm in Perlman's book (Perlman, 1992), it is interesting to note that there was initial resistance to implementing her algorithm, which appeared to be "complex" when compared to simple, fast bridge data forwarding. However, the spanning tree algorithm used control messages, called *Hellos*, that are not processed in real time.

A simple back-of-the-envelope calculation by Tony Lauck related the number of instructions used to process a hello (at most 1000), the rate of hello generation (specified at that time to be once every second), and the number of instructions per second of the Motorola 68000 (around 1 million). Lauck's vision and analysis carried the day, and the spanning tree algorithm was implemented in the final product.

Manufactured at a cost of $1000, the first bridge was initially sold at a markup of around eight, ensuring a handsome profit for DEC when sales initially climbed. In 1986 Mark Kempf was awarded US Patent 4,597,078, titled "Bridge circuit for interconnecting networks." DEC made no money from patent licensing, choosing instead to promote the IEEE 802.1 bridge interconnection standards process.

Together with the idea of self-learning bridges, the spanning tree algorithm has passed into history. Ironically, one of the first customers complained that the bridge did not work correctly; field service later determined that the customer had connected two bridge ports to the same Ethernet, and the spanning tree had (rightly) turned the bridge off! While features like autoconfigurability and provable fault tolerance have only recently been added to Internet protocols, they were part of the bridge protocols in the 1980s.

The success of Ethernet bridges led to proposals for several other types of bridges connecting other local area networks and even wide area bridges. The author even remembers working with John Hart (who went on to become CTO of 3Com) and Fred Baker (who went on to become a Cisco Fellow) on building satellite bridges that could link geographically distributed sites. While some of the initital enthusiasm to extend bridges to supplant routers was somewhat extreme, bridges found their most successful niche in cheaply interconnecting similar local area networks at wire speeds.

[4] This allows a bridged Ethernet to have only 8000 stations. While this is probably sufficient for most customer sites, later bridge implementations raised this figure to 16K and even 64K.

However, after the initial success of 10-Mbps Ethernet bridges, engineers at DEC began to worry about bridging higher-speed LANs. In particular, DEC decided, perhaps unwisely, to concentrate their high-speed interconnect strategy around 100-Mbps FDDI token rings (UNH Inter Operability Lab, 2001). Thus in the early 1990s engineers at DEC and other companies began to worry about building a bridge to interconnect two 100-Mpbs FDDI rings. Could wire speed forwarding, and especially exact-match lookups, be made 10 times faster?

10.3 Challenge 3: scaling lookups to higher speeds

First, let's understand why binary search forwarding *does not* scale to FDDI speeds. Binary search takes $\log_2 N$ memory accesses to look up a bridge database, where N is the size of the database. As bridges grew popular, marketing feedback indicated that the database size needed to be increased from 8 K to 64 K. Thus using binary search, each search would take 16 memory accesses. Doing a search for the source and destination addresses using 100-nanosecond DRAM would then take 3.2 microsecond.

Unlike Ethernet, where small packets are padded to ensure a minimum size of 64 bytes, a minimum-size packet consisting of FDDI, routing, and transport protocol headers could be as small as 40 bytes. Given that a 40-byte packet can be received in 3.2 microsecond at 100 Mbps, two binary search lookups would use up all of the packet-processing budget for a single link, leaving no time for other chores, such as inserting and removing from link chip queues.

One simple approach to meet the challenge of wire speed forwarding is to retain binary search but to use faster hardware (**P5**). In particular, faster SRAM (Chapter 2) could be used to store the database. Given a factor of 5–10 decrease in memory access time using SRAM in place of DRAM, binary search will easily scale to wire speed FDDI forwarding.

However, this approach is unsatisfactory for two reasons. First, it is more expensive because SRAM is more expensive than DRAM. Second, using faster memory gets us lookups at FDDI speeds but will not work for the next speed increment (e.g., Gigabit Ethernet). What is needed is a way to reduce the number of memory accesses associated with a lookup so that bridging can scale with link technology. Of the two following approaches to bridge-lookup scaling, one is based on hashing and the other on hardware parallelism.

10.3.1 Scaling via hashing

In the 1990s DEC decided to build a fast crossbar switch connecting up to 32 links, called the Gigaswitch (Souza et al., 1994). The switch-arbitration algorithms used in this switch will be described in Chapter 13. This chapter concentrates on the bridge-lookup algorithms used in the Gigaswitch. The vision of the original designers, Bob Simcoe and Bob Thomas, was to have the Gigaswitch be a switch connecting point-to-point FDDI links without implementing bridge forwarding and learning. Bridge lookups were considered to be too complex at 100-Mbps speeds.

Into the development arena strode a young software designer who changed the product direction. Barry Spinney, who had implemented an Ada compiler in his last job, was determined to do hardware design at DEC. Barry suggested that the Gigaswitch be converted to a bridge interconnecting FDDI local area networks. To do so, he proposed designing an FDDI-to-Gigaswitch network controller (FGC)

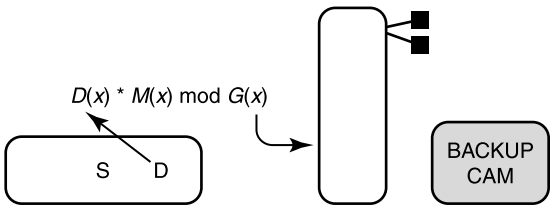

FIGURE 10.3

Gigaswitch hashing uses a hash function with a programmable multiplier, a small, balanced binary tree in every hash bucket, and a backup CAM to hold the rare case of entries that result in more than seven collisions.

chip on the line cards that would implement a hashing-based algorithm for lookups. The Gigaswitch article (Souza et al., 1994) states that each bridge lookup makes at most four reads from memory.

Now, every student of algorithms (Cormen et al., 1990) knows that hashing, on average, is much faster (constant time) than binary search (logarithmic time). However, the same student also knows that hashing is much slower in the worst case, potentially taking linear time because of collisions. How, then, can the Gigaswitch hash lookups claim to take at most four reads to memory in the worst case even for bridge databases of size 64K, whereas binary search would require 16 memory accesses?

The Gigaswitch trick has its roots in an algorithmic technique (**P15**) called *perfect hashing* (Dietzfelbinger et al., 1988; Belazzougui et al., 2009; Limasset et al., 2017). The idea is to use a parameterized hash function, where the hash function can be changed by varying some parameters. Then appropriate values of the parameters can be precomputed (**P2a**) to obtain a hash function such that the worst-case number of collisions is small and bounded.

While finding such a good hash function may take (in theory) a large amount of time, this is a good trade-off because this new station's addresses do not get added to local area networks at a very rapid rate. On the other hand, once the hash function has been picked, lookup can be done at wire speeds.

Specifically, the Gigaswitch hash function treats each 48-bit address as a 47-degree polynomial in the Galois field of order 2, GF(2). While this sounds impressive, this is the same arithmetic used for calculating CRCs; it is identical to ordinary polynomial arithmetic, except that all additions are done mod 2. A hashed address is obtained by the equation $A(X) * M(X) \mod G(X)$, where $G(X)$ is the irreducible polynomial $X^{48} + X^{36} + X^{25} + X^{10} + 1$, $M(X)$ is a nonzero, 47-degree programmable hash multiplier, and $A(X)$ is the address expressed as a 47-degree polynomial.

The hashed address is 48 bits. The bottom 16 bits of the hashed address is then used as an index into a 64K-entry hash table. Each hash table entry [see Fig. 10.3 as applied to the destination address lookup, with $D(x)$ being used in place of $A(x)$] points to the root of a balanced binary tree of height at most 3. The hash function has the property that it suffices to use only the remaining high-order 32 bits of the hashed address to disambiguate collided keys.

Thus the binary tree is sorted by these 32-bit values, instead of the original 48-bit keys. This saves 16 bits to be used for associated lookup information. Thus any search is guaranteed to take no more than four memory accesses, one to lookup the hash table and three more to navigate a height-3 binary tree.

It turns out that picking the multiplier is quite easy in practice. The coefficients of $M(x)$ are picked randomly. Having picked $M(x)$, it sometimes happens that a few buckets have more than seven colliding

addresses. In such a case these entries are stored in a small hardware lookup database called a content addressable memory or CAM (studied in more detail in Chapter 11).

The CAM lookup occurs in parallel with the hash lookup. Finally, in the extremely rare case when several dozen addresses are added to the CAM (say, when new station addresses are learned that cause collisions), the central processor initiates a rehashing operation and distributes the new hash function to the line cards. It is perhaps ironic that rehashing occurred so rarely in practice that one might worry whether the rehashing code was adequately tested!

The Gigaswitch became a successful product, allowing up to 22 FDDI networks to be bridged together with other link technologies, such as ATM. Barry Spinney was assigned US patent 5,920,900, "Hash-based translation method and apparatus with multiple-level collision resolution." While techniques based on perfect hashing (Dietzfelbinger et al., 1988) have been around for a while in the theoretical community, Spinney's contribution was to use a pragmatic version of the perfect hashing idea for high-speed forwarding.

10.3.2 Using hardware parallelism

Techniques based on perfect hashing do not completely provide worst-case guarantees. While they do provide worst-case *search* times of three to four memory accesses, they cannot guarantee worst-case *update* times. It is conceivable that an update takes an unpredictably long time while the software searches for a hash function with the specified bound on the number of collisions.

One can argue that exactly the same guarantees are provided every moment by millions of Ethernets around the world and that nondeterministic update times are far preferable to nondeterministic search times. However, proving that long update times are rare in practice requires either considerable experimentation or good analysis. This makes some designers uncomfortable. It leads to a preference for search schemes that have bounded worst-case search and update times.

An alternate approach is to apply hardware parallelism (**P5**) to a deterministic scheme such as binary search. Binary search has deterministic search and update times; its only problem is that search takes a logarithmic number of memory accesses, which is too slow. We can get around this difficulty by *pipelining* binary search to increase lookup throughput (number of lookups per second) without improving lookup latency. This is illustrated in Fig. 10.4.

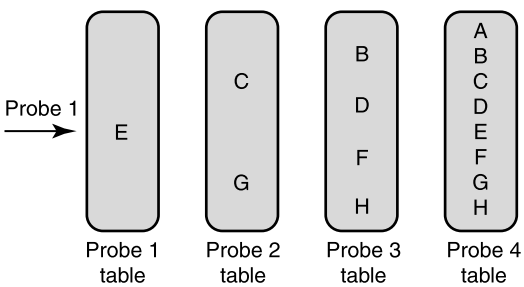

FIGURE 10.4

Pipeling binary search for a database with keys *A* through *H*.

The idea is to have a logarithmic number of processing stages, each with its own memory array. In Fig. 10.4 the keys are the characters A through H. The first array has only the root of the trie, the median element E. The second array corresponds to the quartile and third quartile elements C and G, which are the possible keys at the second probe of binary search, and so on. Search keys enter from the left and progress from stage to stage, carrying a pointer that identifies which key in the corresponding stage memory must be compared to the search key. The lookup throughput is nearly one per memory access because there can be multiple concurrent searches progressing through the stages in order.

Although the figure shows the elements in, say, Stage 2, C and G, as being separated by their spacing in the original table, they can be packed together to save memory in the stages. Thus the overall memory across all stages becomes equal to the memory in a nonpipelined implementation. Indexing into each stage memory becomes slightly more tricky.

Assume Stage i has passed a pointer j to Stage $j + 1$ along with search key S. Stage $j + 1$ compares the search key S to its jth array entry. If the answer is equal, the search is finished but continues flowing through the pipeline with no more changes. If the search key is smaller, the search key is passed to stage $i + 1$ with the pointer $j0$ (i.e., j concatenated with bit 0); if the search key is larger, the pointer passed is $j1$. For example, if the key searched for is F, then the pointer becomes 1 when entering Stage 2 and becomes 10 when entering Stage 3.

The author first heard of this idea from Greg Waters, who later went on to implement IP lookups for the core router company Avici. While the idea looks clever and arcane, there is a much simpler way of understanding the final solution. Computer scientists are well aware of the notion of a binary search tree (Cormen et al., 1990). Any binary search table can be converted into a fully balanced binary search tree by making the root the median element, and so on, along the lines of Fig. 10.4. Any tree is trivially pipelined by height, with nodes of height i being assigned to Stage i.

The only problem with a binary search tree, as opposed to a table, is the extra space required for pointers to children. However, it is well known that for a full binary search tree, such as a heap (Cormen et al., 1990), the pointers can be implicit and can be calculated based on the history of comparisons, as shown earlier. The upshot is that a seemingly abstruse trick can be seen as the combination of three simple and well-known facts from theoretical computer science.

10.3.3 The d-left approach

Now we introduce d-left, the state of art solution (Broder and Mitzenmacher, 2001) to the exact-match problem that has been used in Cisco switch products. The objective of this approach is similar to that of perfect hashing: to minimize the maximum number of objects hashed into any bucket in a hash table. However, when a random hash function is used, this maximum is known to be much larger than the average (Gonnet, 1981). More precisely, when n objects are hashed into n buckets, the expectation of this maximum is $(1 + o(1)) \log n / \log \log n$ (with high probability). As explained earlier, the objective of perfect hashing is to reduce this maximum to a small number (say 2 or 3) so that every bucket can fit into a memory block, using a hash function that is perfect for the set of keys to be inserted into the hash table; as a result, much precomputation is needed to find such a perfect hash function.

d-left is a variation of a slightly older idea called d-random that is now widely known as "the power of d choices" (Mitzenmacher, 1996). The original idea of d-random is to use $d > 1$ random hash functions (say $h_1, h_2, \cdots, h_d$) instead of one. Given a hash key x, there are d candidate hash buckets

into which x can be inserted: buckets indexed by $h_1(x)$, $h_2(x)$, $\cdots$, $h_d(x)$ respectively. Among them, x is inserted into the bucket that is least occupied; ties are broken randomly (which gives the d-random scheme its name). The objective of such a greedy insertion strategy is to keep the number of objects inside each hash bucket as even as possible across the hash table, and as a result, to make the maximum number of objects in any bucket as small as possible. Indeed, it was shown in Broder and Mitzenmacher (2001) that, in d-random, given the same workload above of inserting n objects into a hash table with n buckets, the expectation of this maximum number is reduced to $(\log\log n/\log d) + O(1)$, from $(1 + o(1))\log n/\log\log n$ when only a single random hash function is used.

There are only two slight differences between d-left and d-random. First, in d-left, the hash table is partitioned into d equal-sized (in number of buckets) logically independent subtables, each of which contains n/d buckets (with indices $1, 2, \cdots, n/d$); in comparison, there is no such partitioning in d-random. These d subtables are numbered $1, 2, \cdots, d$ and arranged from left to right, so that subtable 1 is the leftmost, subtable 2 is the second leftmost, and so on. When an object x is to be inserted, each of the d hash functions h_1, h_2, $\cdots$, h_d maps x to a hash value in the range $[1, n/d]$. The d candidate buckets where x can be inserted are the bucket indexed by $h_1(x)$ in subtable 1, the bucket indexed by $h_2(x)$ in the subtable 2, $\ldots$, and the bucket indexed by $h_d(x)$ in subtable d. As in d-random, among these d buckets, the one that is least occupied is where x should be inserted.

Second, in d-left, a tie is broken in a very different way. In d-left if there are more than one least occupied buckets among the d buckets where an object x should be inserted into, then x will be inserted into the leftmost one among them (i.e., the one in the leftmost subtable). This tie-breaking strategy of "going as left as possible" gives d-left its name.

It has been shown that this tie-breaking strategy outperforms, in terms of resulting in a stochastically smaller maximum (occupancy) number, other tie-breaking strategies such as the standard strategy of breaking ties (uniformly) randomly. This finding is perhaps counterintuitive, or even surprising, to some readers. In particular, how can an asymmetric strategy of "always going left" when there is a tie beat a symmetric strategy of breaking ties randomly? We refer readers to Broder and Mitzenmacher (2001) for an intuitive answer to this question, which is still quite subtle. Note that this "going left" tie-breaking strategy performs better only when the performance metric is the maximum occupancy of any bucket. For example, if the performance metric is instead the maximum total occupancy (number of objects in) of any subtable, the random tie-breaking strategy performs better.

Finally, we provide a brief comparison between d-left and perfect hashing. In practice, d-left can achieve a similar maximum occupancy (of any bucket) as perfect hashing. In doing so, d-left does not have to recompute the hash function from time to time (when the set of MAC addresses changes), whereas perfect hashing does. As this recomputation can take a long time for a large set of MAC addresses (e.g., minutes as reported in Broder and Mitzenmacher, 2001), perfect hashing may not be suitable for a LAN environment where the set of MAC addresses "attached" to a switch changes frequently, such as in a campus WiFi network. Compared to perfect hashing, the only obvious disadvantage of d-left is that, when searching for an object x, all the d buckets where x may possibly appear have to be probed. However, since these d buckets belong to logically independent subtables, the probing of these d buckets can be performed in a parallel or pipelined manner (**P5a**), if these d subtables are put in physically independent memory or cache modules.

10.4 Summary

This chapter on exact-match lookups is written as a story, the story of bridging. Three morals can be drawn from this story.

First, bridging was a direct response to the challenge of efficiently extending Ethernets without using routers or repeaters; wire speed forwarding was a direct response to the problem of potentially losing important packets in a flood of less important packets. At the risk of sounding like a self-help book, I hold that challenges are best regarded as opportunities and not as annoyances. The mathematician Felix Klein (Bell, 1986) used to say, "You must always have a problem; you may not find what you were looking for but you will find something interesting on the way." For example, it is clear that the main reason bridges were invented, that is, the lack of high-performance multiprotocol routers, is *not* the reason bridges are still useful today.

This brings us to the second moral. Today it is clear that bridges will never displace routers because of their lack of scalability using flat Ethernet addresses, lack of shortest-cost routing, etc. However, they remain interesting today because bridges are interconnect devices with better cost for performance and higher flexibility than routers for interconnecting a small number of similar local area networks. Thus bridges still abound in the marketplace, often referred to as *switches*. What many network vendors refer to as a *switch* is a crossbar switch, such as the Gigaswitch, that is capable of bridging on every interface. A few new features, notably virtual LANs (VLANs) (Perlman, 1992), have been added. But the core idea remains the same.

Third, the *techniques* introduced by the first bridge have deeply influenced the next generation of interconnect devices, from core routers to Web switches. Recall that Roger Bannister, who first broke the 4-minute-mile barrier, was followed in a few months by several others. In the same way, the first Ethernet bridge was quickly followed by many other wire speed bridges. Soon the idea began to flow to routers as well. Other important concepts introduced by bridges include the use of memory references as a metric, the notion of trading update time for faster lookups, and the use of minimal hardware speedups. All these ideas carry over into the study of router lookups in the next chapter.

In conclusion, the challenge of building the first bridge stimulated creative actions that went far beyond the first bridge. While wire speed router designs are fairly commonplace today, it is perhaps surprising that there are products still being announced that claim gigabit wire speed processing rates for such abstruse networking tasks as encryption and even XML transformations.

10.5 Exercise

1. **ARP Caches:** Another example of an exact-match lookup is furnished by ARP (address resolution protocol) caches in a router or endnode. In an Internet router when a packet first arrives at a destination, the router must store the packet and send an ARP request to the Ethernet containing the packet. The ARP request is broadcast to all endnodes on the Ethernet and contains the IP address of the destination. When the destination responds with an ARP reply containing the Ethernet address of the destination, the router stores the mapping in an ARP table and sends the stored data packet, with the destination Ethernet address filled in.

   - What lookup algorithms can be used for ARP caches?
   - Why might the task of storing data packets awaiting data translation result in packet reordering?

- Some router implementations get around the reordering problem by dropping all data packets that arrive to find that the destination address is not in the ARP table (however, the ARP request is sent out). Explain the pros and cons of such a scheme.

2. **Using CAM to Absorb Overflows From a Hash Table:** Suppose 1 million nodes are hashed into a hash table that contains 1 million buckets. On average, what is the total number of overflows if each bucket can hold at most 4 nodes (say in a memory line)? Calculating this number will help us determine the right size for the CAM. Here, you may assume that the hash function is strictly uniform (across the 1 million indices/buckets). You may also approximate the Binomial random variable you will encounter in this case by a Poisson random variable (since 1 million is a large enough number).