From Kleinberg & Tardos Algorithm Design.

(13.43) *Let $X, X_1, X_2, \ldots, X_n$ and μ be as defined above, and assume that $\mu \leq E[X]$. Then for any $1 > \delta > 0$, we have*

$$\Pr[X < (1-\delta)\mu] < e^{-\frac{1}{2}\mu\delta^2}.$$

The proof of (13.43) is similar to the proof of (13.42), and we do not give it here. For the applications that follow, the statements of (13.42) and (13.43), rather than the internals of their proofs, are the key things to keep in mind.

13.10 Load Balancing

In Section 13.1, we considered a distributed system in which communication among processes was difficult, and randomization to some extent replaced explicit coordination and synchronization. We now revisit this theme through another stylized example of randomization in a distributed setting.

The Problem

Suppose we have a system in which m jobs arrive in a stream and need to be processed immediately. We have a collection of n identical processors that are capable of performing the jobs; so the goal is to assign each job to a processor in a way that balances the workload evenly across the processors. If we had a central controller for the system that could receive each job and hand it off to the processors in round-robin fashion, it would be trivial to make sure that each processor received at most $\lceil m/n \rceil$ jobs—the most even balancing possible.

But suppose the system lacks the coordination or centralization to implement this. A much more lightweight approach would be to simply assign each job to one of the processors uniformly at random. Intuitively, this should also balance the jobs evenly, since each processor is equally likely to get each job. At the same time, since the assignment is completely random, one doesn't expect everything to end up perfectly balanced. So we ask: How well does this simple randomized approach work?

Although we will stick to the motivation in terms of jobs and processors here, it is worth noting that comparable issues come up in the analysis of hash functions, as we saw in Section 13.6. There, instead of assigning jobs to processors, we're assigning elements to entries in a hash table. The concern about producing an even balancing in the case of hash tables is based on wanting to keep the number of collisions at any particular entry relatively small. As a result, the analysis in this section is also relevant to the study of hashing schemes.

Analyzing a Random Allocation

We will see that the analysis of our random load balancing process depends on the relative sizes of m, the number of jobs, and n, the number of processors. We start with a particularly clean case: when $m = n$. Here it is possible for each processor to end up with exactly one job, though this is not very likely. Rather, we expect that some processors will receive no jobs and others will receive more than one. As a way of assessing the quality of this randomized load balancing heuristic, we study how heavily loaded with jobs a processor can become.

Let X_i be the random variable equal to the number of jobs assigned to processor i, for $i = 1, 2, \ldots, n$. It is easy to determine the expected value of X_i: We let Y_{ij} be the random variable equal to 1 if job j is assigned to processor i, and 0 otherwise; then $X_i = \sum_{j=1}^{n} Y_{ij}$ and $E[Y_{ij}] = 1/n$, so $E[X_i] = \sum_{j=1}^{n} E[Y_{ij}] = 1$. But our concern is with how far X_i can deviate above its expectation: What is the probability that $X_i > c$? To give an upper bound on this, we can directly apply (13.42): X_i is a sum of independent 0-1-valued random variables $\{Y_{ij}\}$; we have $\mu = 1$ and $1 + \delta = c$. Thus the following statement holds.

(13.44)

$$\Pr[X_i > c] < \left(\frac{e^{c-1}}{c^c}\right).$$

In order for there to be a small probability of *any* X_i exceeding c, we will take the Union Bound over $i = 1, 2, \ldots, n$; and so we need to choose c large enough to drive $\Pr[X_i > c]$ down well below $1/n$ for each i. This requires looking at the denominator c^c in (13.44). To make this denominator large enough, we need to understand how this quantity grows with c, and we explore this by first asking the question: What is the x such that $x^x = n$?

Suppose we write $\gamma(n)$ to denote this number x. There is no closed-form expression for $\gamma(n)$, but we can determine its asymptotic value as follows. If $x^x = n$, then taking logarithms gives $x \log x = \log n$; and taking logarithms again gives $\log x + \log\log x = \log\log n$. Thus we have

$$2\log x > \log x + \log\log x = \log\log n > 766 \log x,$$

and, using this to divide through the equation $x \log x = \log n$, we get

$$\frac{1}{2}x \leq \frac{\log n}{\log\log n} \leq x = \gamma(n).$$

Thus $\gamma(n) = \Theta\left(\frac{\log n}{\log\log n}\right)$.

Now, if we set $c = e\gamma(n)$, then by (13.44) we have

$$\Pr[X_i > c] < \left(\frac{e^{c-1}}{c^c}\right) < \left(\frac{e}{c}\right)^c = \left(\frac{1}{\gamma(n)}\right)^{e\gamma(n)} < \left(\frac{1}{\gamma(n)}\right)^{2\gamma(n)} = \frac{1}{n^2}.$$

Thus, applying the Union Bound over this upper bound for $X_1, X_2, \ldots, X_n$, we have the following.

(13.45) *With probability at least $1 - n^{-1}$, no processor receives more than $e\gamma(n) = \Theta\left(\frac{\log n}{\log\log n}\right)$ jobs.*

With a more involved analysis, one can also show that this bound is asymptotically tight: with high probability, some processor actually receives $\Omega\left(\frac{\log n}{\log\log n}\right)$ jobs.

So, although the load on some processors will likely exceed the expectation, this deviation is only logarithmic in the number of processors.

Increasing the Number of Jobs We now use Chernoff bounds to argue that, as more jobs are introduced into the system, the loads "smooth out" rapidly, so that the number of jobs on each processor quickly become the same to within constant factors.

Specifically, if we have $m = 16n \ln n$ jobs, then the expected load per processor is $\mu = 16 \ln n$. Using (13.42), we see that the probability of any processor's load exceeding $32 \ln n$ is at most

$$\Pr[X_i > 2\mu] < \left(\frac{e}{4}\right)^{16 \ln n} < \left(\frac{1}{e^2}\right)^{\ln n} = \frac{1}{n^2}.$$

Also, the probability that any processor's load is below $8 \ln n$ is at most

$$\Pr\left[X_i < \frac{1}{2}\mu\right] < e^{-\frac{1}{2}(\frac{1}{2})^2(16 \ln n)} = e^{-2 \ln n} = \frac{1}{n^2}.$$

Thus, applying the Union Bound, we have the following.

(13.46) *When there are n processors and $\Omega(n \log n)$ jobs, then with high probability, every processor will have a load between half and twice the average.*

13.11 Packet Routing

We now consider a more complex example of how randomization can alleviate contention in a distributed system—namely, in the context of *packet routing*.

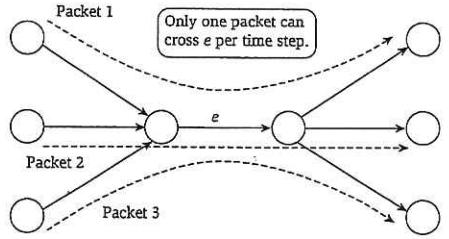

Figure 13.3 Three packets whose paths involve a shared edge e.

The Problem

Packet routing is a mechanism to support communication among nodes of a large network, which we can model as a directed graph $G = (V, E)$. If a node s wants to send data to a node t, this data is discretized into one or more *packets*, each of which is then sent over an s-t path P in the network. At any point in time, there may be many packets in the network, associated with different sources and destinations and following different paths. However, the key constraint is that a single edge e can only transmit a single packet per time step. Thus, when a packet p arrives at an edge e on its path, it may find there are several other packets already waiting to traverse e; in this case, p joins a *queue* associated with e to wait until e is ready to transmit it. In Figure 13.3, for example, three packets with different sources and destinations all want to traverse edge e; so, if they all arrive at e at the same time, some of them will be forced to wait in a queue for this edge.

Suppose we are given a network G with a set of packets that need to be sent across specified paths. We'd like to understand how many steps are necessary in order for all packets to reach their destinations. Although the paths for the packets are all specified, we face the algorithmic question of timing the movements of the packets across the edges. In particular, we must decide when to release each packet from its source, as well as a *queue management policy* for each edge e—that is, how to select the next packet for transmission from e's queue in each time step.

It's important to realize that these *packet scheduling* decisions can have a significant effect on the amount of time it takes for all the packets to reach their destinations. For example, let's consider the tree network in Figure 13.4, where there are nine packets that want to traverse the respective dotted paths up the tree. Suppose all packets are released from their sources immediately, and each edge e manages its queue by always transmitting the packet that is

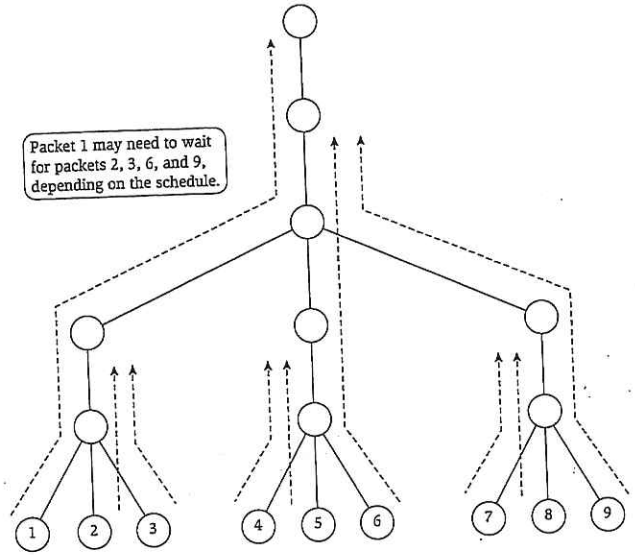

Figure 13.4 A case in which the scheduling of packets matters.

closest to its destination. In this case, packet 1 will have to wait for packets 2 and 3 at the second level of the tree; and then later it will have to wait for packets 6 and 9 at the fourth level of the tree. Thus it will take nine steps for this packet to reach its destination. On the other hand, suppose that each edge e manages its queue by always transmitting the packet that is farthest from its destination. Then packet 1 will never have to wait, and it will reach its destination in five steps; moreover, one can check that every packet will reach its destination within six steps.

There is a natural generalization of the tree network in Figure 13.4, in which the tree has height h and the nodes at every other level have k children. In this case, the queue management policy that always transmits the packet nearest its destination results in some packet requiring $\Omega(hk)$ steps to reach its destination (since the packet traveling farthest is delayed by $\Omega(k)$ steps at each of $\Omega(h)$ levels), while the policy that always transmits the packet farthest from its destination results in all packets reaching their destinations within $O(h+k)$ steps. This can become quite a large difference as h and k grow large.

Schedules and Their Durations Let's now move from these examples to the question of scheduling packets and managing queues in an arbitrary network G. Given packets labeled $1, 2, \ldots, N$ and associated paths $P_1, P_2, \ldots, P_N$, a *packet schedule* specifies, for each edge e and each time step t, which packet will cross edge e in step t. Of course, the schedule must satisfy some basic consistency properties: at most one packet can cross any edge e in any one step; and if packet i is scheduled to cross e at step t, then e should be on the path P_i, and the earlier portions of the schedule should cause i to have already reached e. We will say that the *duration* of the schedule is the number of steps that elapse until every packet reaches its destination; the goal is to find a schedule of minimum duration.

What are the obstacles to having a schedule of low duration? One obstacle would be a very long path that some packet must traverse; clearly, the duration will be at least the length of this path. Another obstacle would be a single edge e that many packets must cross; since each of these packets must cross e in a distinct step, this also gives a lower bound on the duration. So, if we define the *dilation* d of the set of paths $\{P_1, P_2, \ldots, P_N\}$ to be the maximum length of any P_i, and the *congestion* c of the set of paths to be the maximum number that have any single edge in common, then the duration is at least $\max(c, d) = \Omega(c+d)$.

In 1988, Leighton, Maggs, and Rao proved the following striking result: Congestion and dilation are the only obstacles to finding fast schedules, in the sense that there is always a schedule of duration $O(c+d)$. While the statement of this result is very simple, it turns out to be extremely difficult to prove; and it yields only a very complicated method to actually *construct* such a schedule. So, instead of trying to prove this result, we'll analyze a simple algorithm (also proposed by Leighton, Maggs, and Rao) that can be easily implemented in a distributed setting and yields a duration that is only worse by a logarithmic factor: $O(c + d\log(mN))$, where m is the number of edges and N is the number of packets.

Designing the Algorithm

A Simple Randomized Schedule If each edge simply transmits an arbitrary waiting packet in each step, it is easy to see that the resulting schedule has duration $O(cd)$: at worst, a packet can be blocked by $c-1$ other packets on each of the d edges in its path. To reduce this bound, we need to set things up so that each packet only waits for a much smaller number of steps over the whole trip to its destination.

The reason a bound as large as $O(cd)$ can arise is that the packets are very badly timed with respect to one another: Blocks of c of them all meet at an edge at the same time, and once this congestion has cleared, the same thing happens at the next edge. This sounds pathological, but one should remember that a very natural queue management policy caused it to happen in Figure 13.4. However, it is the case that such bad behavior relies on very unfortunate synchronization in the motion of the packets; so it is believable that, if we introduce some randomization in the timing of the packets, then this kind of behavior is unlikely to happen. The simplest idea would be just to randomly shift the times at which the packets are released from their sources. Then if there are many packets all aimed at the same edge, they are unlikely to hit it all at the same time, as the contention for edges has been "smoothed out." We now show that this kind of randomization, properly implemented, in fact works quite well.

Consider first the following algorithm, which will not quite work. It involves a parameter r whose value will be determined later.

```
Each packet i behaves as follows:
  i chooses a random delay s between 1 and r
  i waits at its source for s time steps
  i then moves full speed ahead, one edge per time step
     until it reaches its destination
```

If the set of random delays were really chosen so that no two packets ever "collided"—reaching the same edge at the same time—then this schedule would work just as advertised; its duration would be at most r (the maximum initial delay) plus d (the maximum number of edges on any path). However, unless r is chosen to be very large, it is likely that a collision will occur somewhere in the network, and so the algorithm will probably fail: Two packets will show up at the same edge e in the same time step t, and both will be required to cross e in the next step.

Grouping Time into Blocks To get around this problem, we consider the following generalization of this strategy: rather than implementing the "full speed ahead" plan at the level of individual time steps, we implement it at the level of contiguous *blocks* of time steps.

```
For a parameter b, group intervals of b consecutive time steps
     into single blocks of time
Each packet i behaves as follows:
  i chooses a random delay s between 1 and r
  i waits at its source for s blocks
  i then moves forward one edge per block,
     until it reaches its destination
```


This schedule will work provided that we avoid a more extreme type of collision: It should not be the case that more than b packets are supposed to show up at the same edge e at the start of the same block. If this happens, then at least one of them will not be able to cross e in the next block. However, if the initial delays smooth things out enough so that no more than b packets arrive at any edge in the same block, then the schedule will work just as intended. In this case, the duration will be at most $b(r+d)$—the maximum number of blocks, $r+d$, times the length of each block, b.

(13.47) *Let $\mathcal{E}$ denote the event that more than b packets are required to be at the same edge e at the start of the same block. If $\mathcal{E}$ does not occur, then the duration of the schedule is at most $b(r+d)$.*

Our goal is now to choose values of r and b so that both the probability $\Pr[\mathcal{E}]$ and the duration $b(r+d)$ are small quantities. This is the crux of the analysis since, if we can show this, then (13.47) gives a bound on the duration.

Analyzing the Algorithm

To give a bound on $\Pr[\mathcal{E}]$, it's useful to decompose it into a union of simpler bad events, so that we can apply the Union Bound. A natural set of bad events arises from considering each edge and each time block separately; if e is an edge, and t is a block between 1 and $r+d$, we let $\mathcal{F}_{et}$ denote the event that more than b packets are required to be at e at the start of block t. Clearly, $\mathcal{E} = \cup_{e,t} \mathcal{F}_{et}$. Moreover, if N_{et} is a random variable equal to the number of packets scheduled to be at e at the start of block t, then $\mathcal{F}_{et}$ is equivalent to the event $[N_{et} > b]$.

The next step in the analysis is to decompose the random variable N_{et} into a sum of independent 0-1-valued random variables so that we can apply a Chernoff bound. This is naturally done by defining X_{eti} to be equal to 1 if packet i is required to be at edge e at the start of block t, and equal to 0 otherwise. Then $N_{et} = \sum_i X_{eti}$; and for different values of i, the random variables X_{eti} are independent, since the packets are choosing independent delays. (Note that X_{eti} and $X_{e't'i}$, where the value of i is the same, would certainly not be independent; but our analysis does not require us to add random variables of this form together.) Notice that, of the r possible delays that packet i can choose, at most one will require it to be at e at block t; thus $E[X_{eti}] \leq 1/r$. Moreover, at most c packets have paths that include e; and if i is not one of these packets, then clearly $E[X_{eti}] = 0$. Thus we have

$$E[N_{et}] = \sum_i E[X_{eti}] \leq \frac{c}{r}.$$

We now have the setup for applying the Chernoff bound (13.42), since N_{et} is a sum of the independent 0-1-valued random variables X_{eti}. Indeed, the quantities are sort of like what they were when we analyzed the problem of throwing m jobs at random onto n processors: in that case, each constituent random variable had expectation $1/n$, the total expectation was m/n, and we needed m to be $\Omega(n \log n)$ in order for each processor load to be close to its expectation with high probability. The appropriate analogy in the case at hand is for r to play the role of n, and c to play the role of m: This makes sense symbolically, in terms of the parameters; it also accords with the picture that the packets are like the jobs, and the different time blocks of a single edge are like the different processors that can receive the jobs. This suggests that if we want the number of packets destined for a particular edge in a particular block to be close to its expectation, we should have $c = \Omega(r \log r)$.

This will work, except that we have to increase the logarithmic term a little to make sure that the Union Bound over all e and all t works out in the end. So let's set

$$r = \frac{c}{q \log(mN)},$$

where q is a constant that will be determined later.

Let's fix a choice of e and t and try to bound the probability that N_{et} exceeds a constant times $\frac{c}{r}$. We define $\mu = \frac{c}{r}$, and observe that $E[N_{et}] \leq \mu$, so we are in a position to apply the Chernoff bound (13.42). We choose $\delta = 2$, so that $(1+\delta)\mu = \frac{3c}{r} = 3q \log(mN)$, and we use this as the upper bound in the expression $\Pr\left[N_{et} > \frac{3c}{r}\right] = \Pr\left[N_{et} > (1+\delta)\mu\right]$. Now, applying (13.42), we have

$$\Pr\left[N_{et} > \frac{3c}{r}\right] < \left[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right]^{\mu} < \left[\frac{e^{1+\delta}}{(1+\delta)^{(1+\delta)}}\right]^{\mu} = \left(\frac{e}{1+\delta}\right)^{(1+\delta)\mu}$$
$$= \left(\frac{e}{3}\right)^{(1+\delta)\mu} = \left(\frac{e}{3}\right)^{3c/r} = \left(\frac{e}{3}\right)^{3q \log(mN)} = \frac{1}{(mN)^z},$$

where z is a constant that can be made as large as we want by choosing the constant q appropriately.

We can see from this calculation that it's safe to set $b = 3c/r$; for, in this case, the event $\mathcal{F}_{et}$ that $N_{et} > b$ will have very small probability for each choice of e and t. There are m different choices for e, and $d + r$ different choice for t, where we observe that $d + r \leq d + c - 1 \leq N$. Thus we have

$$\Pr[\mathcal{E}] = \Pr\left[\bigcup_{e,t} \mathcal{F}_{et}\right] \leq \sum_{e,t} \Pr[\mathcal{F}_{et}] \leq mN \cdot \frac{1}{(mN)^z} = \frac{1}{(mN)^{z-1}},$$

which can be made as small as we want by choosing z large enough.

Our choice of the parameters b and r, combined with (13.44), now implies the following.

(13.48) *With high probability, the duration of the schedule for the packets is $O(c + d \log(mN))$.*

Proof. We have just argued that the probability of the bad event $\mathcal{E}$ is very small, at most $(mN)^{-(z-1)}$ for an arbitrarily large constant z. And provided that $\mathcal{E}$ does not happen, (13.47) tells us that the duration of the schedule is bounded by

$$b(r+d) = \frac{3c}{r}(r+d) = 3c + d \cdot \frac{3c}{r} = 3c + d(3q \log(mN)) = O(c + d \log(mN)).$$

■

13.12 Background: Some Basic Probability Definitions

For many, though certainly not all, applications of randomized algorithms, it is enough to work with probabilities defined over finite sets only; and this turns out to be much easier to think about than probabilities over arbitrary sets. So we begin by considering just this special case. We'll then end the section by revisiting all these notions in greater generality.

Finite Probability Spaces

We have an intuitive understanding of sentences like, "If a fair coin is flipped, the probability of 'heads' is 1/2." Or, "If a fair die is rolled, the probability of a '6' is 1/6." What we want to do first is to describe a mathematical framework in which we can discuss such statements precisely. The framework will work well for carefully circumscribed systems such as coin flips and rolls of dice; at the same time, we will avoid the lengthy and substantial philosophical issues raised in trying to model statements like, "The probability of rain tomorrow is 20 percent." Fortunately, most algorithmic settings are as carefully circumscribed as those of coins and dice, if perhaps somewhat larger and more complex.

To be able to compute probabilities, we introduce the notion of a *finite probability space*. (Recall that we're dealing with just the case of finite sets for now.) A finite probability space is defined by an underlying *sample space* Ω, which consists of the possible *outcomes* of the process under consideration. Each point i in the sample space also has a nonnegative *probability mass* $p(i) \geq 0$; these probability masses need only satisfy the constraint that their total sum is 1; that is, $\sum_{i \in \Omega} p(i) = 1$. We define an *event* $\mathcal{E}$ to be any subset of