**COMPUTER NETWORKS-1**

**THE CHANNEL ALLOCATION PROBLEM**

The central theme of this chapter is how to allocate a single broadcast channel among competing users. The channel might be a portion of the wireless spectrum in a geographic region, or a single wire or optical fiber to which multiple nodes are connected. It does not matter. In both cases, the channel connects each user to all other users and any user who makes full use of the channel interferes with other users who also wish to use the channel. We will first look at the shortcomings of static allocation schemes for bursty traffic. Then, we will lay out the key assumptions used to model the dynamic schemes that we examine in the following sections.

**4.1.1 Static Channel Allocation**

The traditional way of allocating a single channel, such as a telephone trunk, among multiple competing users is to chop up its capacity by using one of the multiplexing schemes we described in Sec. 2.5, such as FDM (Frequency Division Multiplexing). If there are *N* users, the bandwidth is divided into *N* equal-sized portions, with each user being assigned one portion. Since each user has a private frequency band, there is now no interference among users. When there are only a small and constant number of users, each of which has a steady stream or a heavy load of traffic, this division is a simple and efficient allocation mechanism. A wireless example is FM radio stations. Each station gets a portion of the FM band and uses it most of the time to broadcast its signal. However, when the number of senders is large and varying or the traffic is bursty, FDM presents some problems. If the spectrum is cut up into *N* regions and fewer than *N* users are currently interested in communicating, a large piece of valuable spectrum will be wasted. And if more than *N* users want to communicate, some of them will be denied permission for lack of bandwidth, even if some of the users who have been assigned a frequency band hardly ever transmit or receive anything.

Even assuming that the number of users could somehow be held constant at *N*,

dividing the single available channel into some number of static sub channels is inherently inefficient. The basic problem is that when some users are quiescent, their bandwidth is simply lost. They are not using it, and no one else is allowed to use it either. A static allocation is a poor fit to most computer systems, in which data traffic is extremely bursty, often with peak traffic to mean traffic ratios of 1000:1. Consequently, most of the channels will be idle most of the time. The poor performance of static FDM can easily be seen with a simple queueing theory calculation. Let us start by finding the mean time delay, *T*, to send a frame onto a channel of capacity *C* bps. We assume that the frames arrive randomly with an average arrival rate of frames/sec, and that the frames vary in length with an average length of 1*/*bits. With these parameters, the service rate of the channel is *C* frames/sec. A standard queueing theory result is

**

(For the curious, this result is for an ‘‘M/M/1’’ queue. It requires that the randomness of the times between frame arrivals and the frame lengths follow an exponential distribution, or equivalently be the result of a Poisson process.)

In our example, if *C* is 100 Mbps, the mean frame length, 1*/*, is 10,000 bits, and the frame arrival rate, , is 5000 frames/sec, then *T* 200 sec. Note that if we ignored the queueing delay and just asked how long it takes to send a 10,000- bit frame on a 100-Mbps network, we would get the (incorrect) answer of 100 sec. That result only holds when there is no contention for the channel. Now let us divide the single channel into *N* independent subchannels, each with capacity *C /N* bps. The mean input rate on each of the sub channels will now be */N.* Re computing *T*, we get

**

The mean delay for the divided channel is *N* times worse than if all the frames were somehow magically arranged orderly in a big central queue. This same result says that a bank lobby full of ATM machines is better off having a single queue feeding all the machines than a separate queue in front of each machine.

Precisely the same arguments that apply to FDM also apply to other ways of statically dividing the channel. If we were to use time division multiplexing (TDM) and allocate each user every *N*th time slot, if a user does not use the allocated slot, it would just lie fallow. The same would hold if we split up the networks physically. Using our previous example again, if we were to replace the 100-Mbps network with 10 networks of 10 Mbps each and statically allocate each user to one of them, the mean delay would jump from 200 sec to 2 msec. Since none of the traditional static channel allocation methods work well at all with bursty traffic, we will now explore dynamic methods.

**Assumptions for Dynamic Channel Allocation**

Before we get to the first of the many channel allocation methods in this chapter, it is worthwhile to carefully formulate the allocation problem. Underlying all the work done in this area are the following five key assumptions:

1. **Independent Traffic**. The model consists of *N* independent **stations** (e.g., computers, telephones), each with a program or user that generatesframes for transmission. The expected number of frames generatedin an interval of length *t* is *t*, where is a constant (the arrivalrate of new frames). Once a frame has been generated, the stationis blocked and does nothing until the frame has been successfullytransmitted.

2. **Single Channel**. A single channel is available for all communication. All stations can transmit on it and all can receive from it. The stations are assumed to be equally capable, though protocols may assign them different roles (e.g., priorities).

3. **Observable Collisions**. If two frames are transmitted simultaneously, they overlap in time and the resulting signal is garbled. This event is called a **collision**. All stations can detect that a collision has occurred. A collided frame must be transmitted again later. No errors other than those generated by collisions occur.

4. **Continuous or Slotted Time**. Time may be assumed continuous, in which case frame transmission can begin at any instant. Alternatively, time may be slotted or divided into discrete intervals (called slots). Frame transmissions must then begin at the start of a slot. A slot may contain 0, 1, or more frames, corresponding to an idle slot, an successful transmission, or a collision, respectively.

5. **Carrier Sense or No Carrier Sense**. With the carrier sense assumption, stations can tell if the channel is in use before trying to use it. No station will attempt to use the channel while it is sensed as busy. If there is no carrier sense, stations cannot sense the channel before trying to use it. They just go ahead and transmit. Only later can they determine whether the transmission was successful.

Some discussion of these assumptions is in order. The first one says that frame arrivals are independent, both across stations and at a particular station, and that frames are generated unpredictably but at a constant rate. Actually, this assumption is not a particularly good model of network traffic, as it is well known that packets come in bursts over a range of time scales (Paxson and Floyd, 1995; and Leland et al., 1994). Nonetheless, **Poisson models**, as they are frequently called, are useful because they are mathematically tractable. They help us analyze protocols to understand roughly how performance changes over an operating range and how it compares with other designs.

The single-channel assumption is the heart of the model. No external ways to communicate exist. Stations cannot raise their hands to request that the teacher call on them, so we will have to come up with better solutions. The remaining three assumptions depend on the engineering of the system, and we will say which assumptions hold when we examine a particular protocol. The collision assumption is basic. Stations need some way to detect collisions if they are to retransmit frames rather than let them be lost. For wired channels, node hardware can be designed to detect collisions when they occur. The stations can then terminate their transmissions prematurely to avoid wasting capacity. This detection is much harder for wireless channels, so collisions are usually inferred after the fact by the lack of an expected acknowledgement frame. It is also possible for some frames involved in a collision to be successfully received, depending on the details of the signals and the receiving hardware. However, this situation is not the common case, so we will assume that all frames involved in a collision are lost. We will also see protocols that are designed to prevent collisions from occurring in the first place. The reason for the two alternative assumptions about time is that slotted time can be used to improve performance. However, it requires the stations to follow a master clock or synchronize their actions with each other to divide time into discrete intervals. Hence, it is not always available. We will discuss and analyze systems with both kinds of time. For a given system, only one of them holds.

Similarly, a network may have carrier sensing or not have it. Wired networks will generally have carrier sense. Wireless networks cannot always use it effectively because not every station may be within radio range of every other station. Similarly, carrier sense will not be available in other settings in which a station cannot communicate directly with other stations, for example a cable modem in which stations must communicate via the cable headed. Note that the word ‘‘carrier’’ in this sense refers to a signal on the channel and has nothing to do with the common carriers (e.g., telephone companies) that date back to the days of the Pony Express.

To avoid any misunderstanding, it is worth noting that no multi-access protocol guarantees reliable delivery. Even in the absence of collisions, the receiver may have copied some of the frame incorrectly for various reasons. Other parts of the link layer or higher layers provide reliability.

**Crash Recovery**

If hosts and routers are subject to crashes or connections are long-lived (e.g., large software or media downloads), recovery from these crashes becomes an issue. If the transport entity is entirely within the hosts, recovery from networkand router crashes is straightforward. The transport entities expect lost segments all the time and know how to cope with them by using retransmissions. A more troublesome problem is how to recover from host crashes. In particular, it may be desirable for clients to be able to continue working when servers crash and quickly reboot. To illustrate the difficulty, let us assume that one host, the client, is sending a long file to another host, the file server, using a simple stop-and-wait protocol. The transport layer on the server just passes the incoming segments to the transport user, one by one. Partway through the transmission, the server crashes. When it comes back up, its tables are reinitialized, so it no longer knows precisely where it was.



In an attempt to recover its previous status, the server might send a broadcast segment to all other hosts, announcing that it has just crashed and requesting that its clients inform it of the status of all open connections. Each client can be in one of two states: one segment outstanding, *S1*, or no segments outstanding, *S0*.

Based on only this state information, the client must decide whether to retransmit the most recent segment.

At first glance, it would seem obvious: the client should retransmit if and only if it has an unacknowledged segment outstanding (i.e., is in state *S1*) when it learns of the crash. However, a closer inspection reveals difficulties with this naive approach. Consider, for example, the situation in which the server’s transport entity first sends an acknowledgement and then, when the acknowledgement has been sent, writes to the application process. Writing a segment onto the output stream and sending an acknowledgement are two distinct events that cannot be done simultaneously.

This problem gets us into the issue of what a so-called end-to-end acknowledgement really means. In principle, the transport protocol is end-to-end and not chained like the lower layers. Now consider the case of a user entering requests for transactions against a remote database. Suppose that the remote transport entity is programmed to first pass segments to the next layer up and then acknowledge. Even in this case, the receipt of an acknowledgement back at the user’s machine does not necessarily mean that the remote host stayed up long enough to actually update the database. A truly end-to-end acknowledgement, whose receipt means that the work has actually been done and lack thereof means that it has not, is probably impossible to achieve. This point is discussed in more detail by Saltzer et al. (1984).

**Exponential backoff**

Exponential backoff is an [algorithm](http://en.wikipedia.org/wiki/Algorithm) that uses [feedback](http://en.wikipedia.org/wiki/Feedback) to multiplicatively decrease the rate of some process, in order to gradually find an acceptable rate.

**Binary exponential backoff / truncated exponential backoff**

In a variety of [computer networks](http://en.wikipedia.org/wiki/Computer_networks), binary exponential backoff or truncated binary exponential backoff refers to an [algorithm](http://en.wikipedia.org/wiki/Algorithm) used to space out repeated [retransmissions](http://en.wikipedia.org/wiki/Retransmission_(data_networks)) of the same block of [data](http://en.wikipedia.org/wiki/Data), often as part of [network congestion avoidance](http://en.wikipedia.org/wiki/Network_congestion_avoidance).

Examples are the retransmission of [frames](http://en.wikipedia.org/wiki/Data_frame) in [carrier sense multiple access with collision avoidance](http://en.wikipedia.org/wiki/Carrier_sense_multiple_access_with_collision_avoidance) (CSMA/CA) and [carrier sense multiple access with collision detection](http://en.wikipedia.org/wiki/Carrier_sense_multiple_access_with_collision_detection) (CSMA/CD) networks, where this algorithm is part of the [channel access](http://en.wikipedia.org/wiki/Media_access_control) method used to send data on these network. In [Ethernet](http://en.wikipedia.org/wiki/Ethernet) networks, the algorithm is commonly used to schedule retransmissions after collisions. The retransmission is delayed by an amount of [time](http://en.wikipedia.org/wiki/Time) derived from the [slot time](http://en.wikipedia.org/wiki/Slot_time) and the number of attempts to retransmit.

After *c* collisions, a random number of slot times between 0 and 2c - 1 are chosen. For the first collision, each sender will wait 0 or 1 slot times. After the second collision, the senders will wait anywhere from 0 to 3 slot times ([inclusive](http://en.wikipedia.org/wiki/Interval_(mathematics))). After the third collision, the senders will wait anywhere from 0 to 7 slot times (inclusive), and so forth. As the number of retransmission attempts increases, the number of possibilities for delay [increases exponentially](http://en.wikipedia.org/wiki/Exponential_growth).

The 'truncated' simply means that after a certain number of increases, the exponentiation stops; i.e. the retransmission timeout reaches a ceiling, and thereafter does not increase any further. For example, if the ceiling is set at *i* = 10 (as it is in the [IEEE 802.3](http://en.wikipedia.org/wiki/IEEE_802.3) CSMA/CD standard), then the maximum delay is 1023 slot times.

Because these delays cause other stations that are sending to collide as well, there is a possibility that, on a busy network, hundreds of people may be caught in a single collision set. Because of this possibility, after 16 attempts at transmission, the process is aborted. [[*Citation needed*](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)]

**An example of an exponential backoff algorithm**

This example is from the [Ethernet](http://en.wikipedia.org/wiki/Ethernet) protocol,[[2]](http://en.wikipedia.org/wiki/Exponential_backoff#cite_note-2) where a sending host is able to know when a collision has occurred (that is, another host has tried to transmit), when it is sending a frame. If both hosts attempted to retransmit as soon as a collision occurred, there would be yet another collision — and the pattern would continue forever. The hosts must choose a random value within an acceptable range to ensure that this situation doesn't happen. An exponential backoff algorithm is therefore used. The figure 51.2μs is used as an example here but it is the slot time for a 10 Mbit/s Ethernet line (see [Slot time](http://en.wikipedia.org/wiki/Slot_time)). However, 51.2μs could be replaced by any positive value, in practice.

1. When a collision first occurs, send a “Jamming signal” to prevent further data being sent.
2. Resend a frame after either 0 seconds or 51.2μs, chosen at random.
3. If that fails, resend the frame after either 0s, 51.2μs, 102.4μs, or 153.6μs.
4. If that still doesn't work, resend the frame after k · 51.2μs, where *k* is a random number between 0 and 23 − 1.
5. In general, after the *c*th failed attempt, resend the frame after k · 51.2μs, where *k* is a random number between 0 and 2*c* − 1.

**Expected backoff**

Given a [uniform distribution](http://en.wikipedia.org/wiki/Uniform_distribution_(discrete)) of backoff times, the [expected](http://en.wikipedia.org/wiki/Expected_value) backoff time is the mean of the possibilities. That is, after *c* collisions, the number of backoff slots is in [0, 1, ..., *N*] where *N* = 2*c* - 1 and the expected backoff time (in slots) is

\frac{1}{N+1}\sum_{i=0}^{N} i.

For example, the expected backoff time for the third (*c* = 3) collision, one could first calculate the maximum backoff time, *N*:

N = 2^c - 1

N = 2^3 - 1 = 8 - 1

N = 7

... and then calculate the mean of the backoff time possibilities:

\operatorname{E}(c) = \frac{1}{N+1}\sum_{i=0}^{N} i

\operatorname{E}(3) = \frac{1}{7+1}\sum_{i=0}^{7} i = \frac{1}{8}(0 + 1 + 2 + 3 + 4 + 5 + 6 + 7) = \frac{28}{8}

\operatorname{E}(3) = 3.5

... obtaining 3.5 as the expected number of back off slots after 3 collisions.

The above derivation is largely unnecessary when you realize that the mean of consecutive integers is equal to the mean of the largest and smallest numbers in the set. That is, the mean of a set*A* of consecutive integers *a*0, *a*1, *a*2, ... *a*m is simply the mean of *a*0 and *a*m or (*a*0 + *a*m) / 2.

When applied to the above problem of finding the expected backoff time, the formula becomes simply:

\operatorname{E}(c) = \frac{2^c - 1}{2}

... or otherwise interpreted as half of the maximum backoff time.