Network Security

Professor Gaurav S. Kasbekar
Department of Electrical Engineering
Indian Institute of Technology, Bombay

Week - 01

Lecture - 04

Review of Basic Concepts and Terminology in Communication Networks: Part 2

Hello, in this lecture, we continue our review of basic concepts and terminology in communication networks. So, there are four basic networking functions. These are routing, reliable data transfer, congestion control, and medium access control. We'll provide a review of each of these four networking functions. This is not an exhaustive list.

For example, security functions are omitted from this list. We'll discuss security functions later on in the rest of the course. We start with routing, which is the first networking function we discussed. Recall that data is transported between end systems via several intermediate routers. So, in this example, this server and this desktop computer, these are end systems, and data is transported between them by several intermediate routers which are shown by these blue icons.

In a network, there is a mesh of interconnected routers, which is shown here by this red highlighting. So, there is a mesh of many interconnected routers. Typically, there are several possible routes between a given pair of end systems. In this example, consider this end system, let's call it A, and this end system, let's call it B. We can see that there are several possible routes from A to B. One route is this. Another route is this, so on and so forth.

So, we can see that even in this small network there are many routes from A to B. The routing problem is to find the best route between each pair of end systems. What do we mean by the best route? We'll define that later on. To study the routing problem, it is useful to abstract it using a graph. A graph consists of a set of nodes N and a set of edges E. These blue icons are the nodes, and these links connecting them are the edges.

So, N is the set of nodes, and E is the set of edges. There is one node corresponding to each router. In this graph, there is one node each corresponding to each router, so, u, v, x, w, y, and z; these are the nodes corresponding to different routers. And there is one edge for

every communication link. So, for example, there is an edge from router u to router v. There is an edge from router v to router w, and so on.

So, these are the set of communication links. So, we represent the routers by nodes and communication links by edges. Now, notice that some numbers are written next to the edges. These are the costs of the corresponding communication links. For example, the cost of the link from w to z is 5.

That is shown here. The cost of this link is 5. So, how do we assign costs to different communication links? The cost can be assigned in different ways depending on what we want to optimize. For example, the cost could always be 1, that is, 1 for every link.

The cost might be proportional to the physical length of the link. It may be equal to the monetary cost of using the link, and so on and so forth. Now, the cost of a path is defined to be the sum of the cost of the links in the path. For example, consider some paths from u to z. The cost of this path from u to v, v to w, and w to z. The cost of the path u, v, w, z; that cost is the sum of the cost of the edges in the path. That is, in this example, 2 + 3 + 5 = 10.

As another example, the cost of this path from u to x, x to y, and y to z, the cost of this path is the following. Cost of the path u, x, y, z is 1 + 2 = 4. So, we can see that there are multiple paths with different costs. Now, the shortest path problem is to find the least-cost path between two nodes, for example, u and z. In this simple example, we can see by inspection that the least-cost path is this one: u, x, y, z. It has a cost of 4. So, this is the shortest path problem, to find the least-cost path between two nodes, for example, u and z, in this example.

So, depending on how we have assigned costs to links, the least-cost path optimizes different objectives. For example, if the cost of every link is 1, then it is straightforward to check that the least-cost path minimizes the number of hops from the source to the destination. If the cost is proportional to the physical length of the link, then the least-cost path minimizes the total propagation delay from the source to the destination. If the cost of a link is equal to the monetary cost of using the link, then the least-cost path minimizes the total monetary cost from the source to the destination. So, depending on what we want to optimize, we assign cost to links in different ways.

Now, how do we find the shortest path in a network efficiently? So, a "Brute Force Solution" is to try out every path from the source to the destination, and identify the path

with the smallest cost. But this requires a very large number of computations, even in small networks. So, there are more efficient algorithms designed for finding shortest paths in a network. The most widely deployed routing algorithms are Dijkstra's algorithm and Bellman-Ford's algorithm.

The popular protocol OSPF is based on Dijkstra's algorithm, and the popular protocol RIP is based on Bellman-Ford's algorithm. Dijkstra's algorithm is a centralized algorithm. First, every router sends the list of all its neighbors, and the cost of the corresponding links to every other router in the network. Now, at a particular router, information about the entire network graph is available. The router then runs Dijkstra's algorithm, which is a centralized algorithm for efficiently computing shortest paths from itself to all the other nodes in the network.

So, this is a centralized algorithm. First, the entire graph topology is collected at a router, and then the router computes shortest paths from itself to all the other nodes in the network. And in contrast, Bellman-Ford's algorithm is a distributed algorithm. Here, all nodes perform some computations, and keep on exchanging messages with their neighbors in parallel. And when the algorithm converges, each node v knows the next hop on the shortest path to every destination u. It does not know the complete path, but the complete path is not required.

Each node only needs to know which is the next hop towards a particular destination u. So, this Bellman-Ford's is a distributed algorithm. We won't discuss the details of these algorithms. You can refer to any networking textbook or algorithms textbook to get a discussion of the details of these algorithms. So, this shows the operation of Bellman-Ford's algorithm in a simple example. This is a graph with these routers, u, v, w, x, y, and z, and the cost of different links are indicated next to the corresponding links.

This shows one example operation of Bellman-Ford's algorithm. These D(v), D(w), D(x), D(y), and D(z); these are estimates of the currently known shortest path from every node to the destination u. So, every node wants to find a shortest path to the destination u. So, in this example, v initializes its path to 1 because there is a direct link from v to u. So v initializes D(v) to 1. There is no direct link from w to u, so, w initializes its distance estimate to u to infinity, and so, do x and y, and z has a link of cost 6, to u, so, it initializes D(z) to 6. So, then v sends a message to w indicating that it has a path of cost 1 to u. Then w finds out that it has a link of cost 1 to v, and from there it can get to u with the cost of 1.

So, hence, w knows that it has a path of cost 2 to u. So, w updates D(w) to 2 and then, w sends a message to x, and x updates its estimate to 3, and so on and so forth.

So, in this way, the algorithm converges to the shortest path cost. So, these final numbers in every column, these are the shortest path from the corresponding node to node u. This is another execution of Bellman-Ford's algorithm. They initialize their distance estimates in the same way as before. Now, z happens to be the first node which sends a message to its neighbor y. Then y updates its distance estimate to 7; 1 + 6 = 7. So then, y happens to send the next message to x, and x updates its distance estimate to 8, and so on and so forth.

So, after convergence, the final distance estimates are the same as in the first case. And this is yet another execution. Here, v happens to be the first node that sends a message to its neighbor w. Then, z happens to be the next node, which sends a message to its neighbor y. So, regardless of the order in which different nodes exchange messages among themselves, the algorithm always converges to the same shortest path cost. This is an example, which illustrates the execution of Bellman-Ford's algorithm. We can see that nodes keep on updating their distance estimates, and they keep on exchanging messages with their neighbors.

When the algorithm converges, each node has an estimate of the shortest path cost from itself to the destination. So, we discussed how to compute shortest path cost. The node to which a node must send a packet to achieve the shortest path cost can also be similarly stored. We'll omit the details. For example, in this example, w knows that it must send its packet to v to reach u by the shortest path, and so on.

So, this Bellman-Ford's algorithm allows us to find the shortest path cost as well as the shortest path themselves. So far, we discussed what is known as unicast routing, which is transmission from a single source to a single destination. Sometimes, we require transmission from a single source to all the other nodes in the network. This is known as a broadcast transmission. It is often used to distribute control information.

For example, if each node broadcasts a list of its neighbors and the associated link costs, then all the nodes in the network know the full topology. And this knowledge can be used for finding the shortest paths and so on. So, we sometimes require to send some information, sometimes a node requires to send some information to all the other nodes in the network. This is known as the broadcast problem. We need an efficient way to broadcast packets.

A packet can be efficiently broadcasted using what is known as a spanning tree. So, in a graph, a spanning tree is defined in this way. If you have a graph G with node set N and edge set E, then a spanning tree is a graph G' = (N, E'). That is, G' has the same node set as the original graph G, and the edge set is a subset of the original edge set E. And G' is connected and it has no cycles. So, in this graph, the set of all links, whether solid or thin, they constitute the original graph G, and the set of thick edges, they constitute the spanning tree.

So, we can see that the spanning tree is connected and it has no cycles. A spanning tree can be used to efficiently broadcast information. Here, A wants to broadcast a packet to every other node in the network. So, first, we identify a spanning tree. So, the links against which there are arrows, they are part of the spanning tree, and the links on which there are no arrows, they are not part of the spanning tree.

So, A sends the message only along the edges of the spanning tree, so, it sends a message along this edge and along this edge. Then, C sends the message along this edge and along this edge, but not along this edge because this edge is not part of the spanning tree, and so on and so forth. So, we can see that exactly one copy of the packet reaches every node. So, a broadcast is accomplished in an efficient manner. So, a broadcast: for using a spanning tree for broadcasting, all the nodes have to first agree on which spanning tree to use.

So, that can only be done if a spanning tree has been identified before. If a sparing tree has not been identified, then a process called control flooding can be used where A sends a copy of the packet to every neighbor, and then every neighbor sends a copy of the packet to every other neighbor, and so on and so forth. So, a copy of the packet reaches all the nodes in the network eventually. And by control flooding we mean that a node, every packet is marked with a particular number called sequence number. So, the first packet might be marked with number 1, second packet number 2, and third packet number 3, and so on.

And if a node receives a packet with the same sequence number as before, then it drops the packet. So, this is to avoid unnecessary transmissions of the same packets over and over again. So, this is known as control flooding. So, that can be used if a spanning tree has not been identified. This concludes our brief review of routing in networks.

Next, we discuss reliable data transfer. So, the reliable data transfer problem is as follows. Suppose a large file needs to be transferred from an end system A to an end system B. The file is broken into a number of chunks called packets, as discussed before. Now, later we'll

discuss layering in networks. Basically, the set of all functions which need to be performed, they are broken down into different components; these are known as layers.

So, the layers in the internet are Application, Transport, Network, Data link, and Physical. So, we'll discuss these layers later. But for now, we only note that the network layer implements routing. And it provides unreliable transfer, that is, the network layer tries its best to transfer information from A to B. But some bit errors are possible. That is, some bits may get flipped from 0 to 1 or from 1 to 0.

Some packets might get lost. Typically, a packet gets lost because a router buffer runs out of space. So, there is no space in the router buffer to store the packet. So, the packet is dropped. Packets may also be reordered.

This happens because packets may take different routes on transit from A to B. So, the network layer provides unreliable transfer. Now, the transport layer must implement reliable transfer using the network layer for unreliable transfer and some additional mechanisms. For example, checksums and retransmissions, which we'll discuss on the next slide. So, the transport layer uses the services of the network layer, namely unreliable transfer, and adds some mechanisms of its own to achieve reliable transfer. That is, bit errors are compensated for, lost packets are retransmitted, and packets are put in the correct order.

So, let's discuss mechanisms that can be used for reliable data transfer. So, if a packet is transferred from A to B, how does B detect whether there are any bit errors in the packet? So, this can be done using a checksum. So, examples of checksums are parity checksum and Hamming code, and so on. So, we add some extra redundant bits to a packet and by examining the checksum, the receiver can find out whether there are errors in the packet or not.

Now, if a receiver receives a packet and the checksum verification succeeds, then how does it confirm that the packet has been correctly received to the sender? So, to confirm, the correct receipt of a data packet: A packet known as an acknowledgement packet is sent from the receiver to the sender. This indicates to the sender that a particular packet has been correctly received. Now, if a packet is corrupted during transit from the sender to the receiver or it is lost during transit from the sender to the receiver, then how does the sender compensate for such packets?

The sender retransmits the same packet to the receiver to compensate for the corrupt or lost packet. Suppose the sender has sent a packet, but it never reached the receiver because it was dropped during transit at a router because of congestion at the router. So, how does the sender detect that a packet has been lost? So, when the sender sends a packet, it starts a timer, which is set to an estimate of the time required for a packet to reach the sender and then for the acknowledgement to come back to the sender. So, the timeout interval is set to an estimate of the time required for a data packet to go from the sender to the receiver and for the acknowledgement to come back from the receiver to the sender.

And if the timer expires and still the acknowledgement has not been received, then the sender knows that the packet may have been lost, and it can resubmit the packet. Now, how does a receiver detect duplicate packets and correctly order the received packets at the receiver? So, this is done by adding sequence numbers in data packets. The first packet might have a sequence number of 1, second packet will have a sequence number of 2, and so on. Third packet has a sequence number of 3, and so on.

So, by looking at the sequence numbers in packets, the receiver can put the received packets in the correct order, even if they are reordered in the network. And if a receiver receives duplicate packets, then it can discard the duplicate copy, and retain only one copy of each packet. So, duplicate packets are sent sometimes because the sender thinks that a packet has been lost but actually the acknowledgement has only been delayed. The packet has not been lost. So, such duplicate packets can be eliminated also using sequence numbers.

This shows an example operation of a reliable data transfer protocol. In this figure on the left, it shows operation with no loss. The sender sends a packet to the receiver and the receiver sends an acknowledgement, then the sender sends the next packet and the receiver sends an acknowledgement, and so on. So here, the first packet has a sequence number of 0, second packet has a sequence number of 1, third packet has a sequence number of 0, and so on. Sequence numbers are modulo 2.

In general, sequence numbers are modulo a larger number. For example, there might be, say, 10,000 different sequence numbers used. So, the sequence numbers go from 0 to 9999. Fine, so this shows a lost packet. The first packet sent by the sender reaches the receiver and it is acknowledged.

Then, the next packet is sent, but it is lost. Then a timer expires at the sender, so, it resends the same packet again and then it is acknowledged, and the next packet is sent, and so on. This figure on the left shows the case of a lost ACK. The first packet is sent, and it is

correctly received and acknowledged by the receiver. Then, the next packet is sent and it is correctly received at the receiver, but the acknowledgement is lost, possibly because of transition in the network.

So, after a timeout happens at the sender, it resends the same packet. So, we can see that it had earlier sent a packet with sequence number 1. It again sends the packet with the same sequence number 1. That is, the sequence number repetition indicates that it is a retransmission of the same packet. So, the receiver sees that the sequence number is the same as before, so, it discards the duplicate because it has already received packet with sequence number 1, but it still sends an acknowledgement because it doesn't know whether the acknowledgement has reached the sender in the previous instance.

So, using the sequence number, the receiver is able to detect and discard a duplicate packet. And then, the next packet is sent by the sender and it is correctly acknowledged. This fourth case shows the case of a premature timeout. The sender sends a packet with sequence number 0; it is correctly acknowledged, then packet 1 is sent, and its acknowledgement is delayed, possibly because of congestion in the network. The timer expires at the sender and it resends the same packet.

The receiver detects a duplicate, and then using a sequence number, it is able to detect that it is the same packet as before. So, it discards the duplicate and sends an acknowledgement, and then, the sender at this point sends the next packet, and so on and so forth. So, these are different examples of the operation of a reliable data transfer protocol. So, reliable data transfer protocols are of the following types: Stop-and-Wait and Pipelined. A stop-and-wait protocol is illustrated on the left, and a pipelined protocol is illustrated on the right.

By stop-and-wait, we mean that the sender sends one packet and then waits for its acknowledgement. Only after receiving its acknowledgement, the sender sends the next packet, and so on and so forth. And in contrast, in a pipelined protocol, the sender keeps on sending new packets even before the acknowledgement of the first packet has been received. So, this is a pipelined protocol. Data transfer happens in parallel on different communication links from the sender to the receiver.

Link utilization is poor in stop-and-wait because, if the round trip time from the sender to the receiver is large, round trip time is the time for the data packet to reach from the sender to the receiver, and for the acknowledgement to come back to the sender. So, if the round trip time is large then the sender has to wait for a long time for the acknowledgement of the first packet, and during this interval, it cannot send new packets to the receiver. So,

pipelined protocol is more efficient than a stop-and-wait protocol. There are different pipelined protocols used in the internet. TCP uses a pipelined protocol.

It sends packets while it is waiting for the acknowledgement of the first of those packets. We use a parameter called "window," which imposes a limit on the number of packets that can be sent without waiting for the acknowledgement of the first of those packets. In this example, the window size is 3, which means that the sender can send at most 3 packets in the interval before the first acknowledgement is received. So, since the window size is 3, the sender sends 3 packets: packet 1, packet 2, and packet 3, and then it gets blocked because the window only allows it to send 3 packets, and then it must wait for the acknowledgement of one of the packets. Now, since the acknowledgement of the first packet is received, now only packets 2 and 3 have been sent and not yet acknowledged.

So, since the window is 3, the sender can send one more packet, say packet 4. So, now packets 2, 3, and 4 are out and not yet acknowledged. So, the sender must wait for an acknowledgement. The sender waits for acknowledgement. It receives acknowledgement 2 at this point.

Then it sends packet 5 and so on and so forth. So, at any time, there can be at most 3 packets, which are sent but not yet acknowledged. So, why do we impose a finite window? Why not allow the sender to just keep on sending packets without waiting for an acknowledgement? So, there are different reasons for a finite window. One reason is to control congestion.

So, if the sender were to keep on sending packets continuously without using any window, then there could be a lot of congestion in the network. In contrast, if we use a finite window and we adapt the window size to the detected congestion level in the network, then that can control congestion in the network. Another reason for using a finite window is to avoid exceeding the buffer space at the receiver. So, suppose the receiver has a buffer space of, say, 1,000 packets. So, if the sender sends more than 1,000 packets to the receiver, then the receiver's buffer will run out of space.

Another reason for using a finite window is to avoid exceeding the buffer space at the receiver. This concludes our discussion of reliable data transfer. In the next lecture, we'll discuss the other networking functions, such as congestion control and medium access control. Thank you.