

Theory of Communication Networks

Gopal Pandurangan
Purdue University

Maleq Khan
Purdue University

June 16, 2008

1 Introduction

Communication networks have become ubiquitous today. The Internet, the global computer communication network that interconnects millions of computers, has become an indispensable part of our everyday life. This chapter discusses theoretical and algorithmic underpinnings of distributed communication networks, focusing mainly on themes motivated by the Internet. The Internet is a distributed wide area communication network that connects a variety of end systems or *hosts* by a network of communication links and packet switches (e.g., routers). A packet switch takes a packet arriving on one of its incoming communication links and forwards that packet on one of its outgoing communication links. From the sending host to the receiving host, the sequence of communication links and packet switches is known as a **route** or a path through the network. Throughout we will use the term node to denote a host (processor) or a packet switch.

The Internet is a complex system, but fortunately it has a layered architecture which is extremely helpful in understanding and analyzing its functionality. This is called as the *network protocol stack* and is organized as follows. The understanding of different layers of the stack allows us to tie the theoretical and algorithmic results that will be discussed to specific functions and protocols in the Internet.

Application Layer: The application layer is closest to the end user. This layer interacts with software applications that implement a communicating component. The layered architecture allows one to create a variety of distributed application protocols running over multiple hosts. The application in one host uses the protocol to exchange packets of data with the application in another host. Some examples of application layer implementations include Telnet, File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP), and the Hypertext Transfer Protocol (HTTP). An application architecture determines how a network application is structured over the various hosts. The traditional application architecture paradigm has been the client-server paradigm. In a client-server architecture, there is an always-on host, called the server, which

services requests from many other hosts, called clients. For example, all the above applications — Web, FTP, Telnet, and email — are client-server based. The last few years has seen the emergence of a new paradigm called *Peer-to-Peer (P2P)* architecture. In P2P architecture, there is no concept of a dedicated, always-on, server. Instead hosts, called as peers, communicate directly and a peer can act both as a client (while requesting information) or as a server (when servicing requests for other peers). Many of today's most popular and traffic-intensive applications such as file distribution (e.g., BitTorrent), file searching (e.g., Gnutella/LimeWire), Internet telephony (e.g., Skype) are P2P based. A key application of P2P is the decentralized searching and sharing of data and resources. The P2P paradigm is inherently scalable as the peers serve the dual role of clients and servers.

Transport Layer: The transport layer segments the data for transport across the network. Generally, the transport layer is responsible for making sure that the data is delivered error-free and in the proper sequence. Flow control (i.e., sender/receiver speed matching) and congestion control (source throttles its transmission rate as a response to network congestion) occurs at the transport layer. TCP and UDP are the two transport protocols of the Internet, with only the former providing flow and congestion control.

Network Layer: The network layer is responsible for breaking data into packets (known as datagrams) and moving the packets from the source to destination. The network layer defines the network address. It includes the Internet Protocol (IP), which defines network addresses, called as **IP addresses**. Because this layer defines the logical network layout, routers can use this layer to determine how to forward packets. Internet's routing protocols are a key part of this layer. The routing protocols help in configuring the *forwarding tables* of the routers which indicates to which of the neighbors a packet is to be forwarded based on its destination address. There are two types of protocols based on whether the routing is made within an Autonomous system (intra-AS routing) or between Autonomous systems (inter-AS routing). **Shortest path routing** is typically used for intra-AS routing. A protocol called *Border Gateway Protocol (BGP)* is used for inter-AS routing.

Link Layer: Link layer's functionality is to move data from one node to an adjacent node over a single link on the path from the source to the destination host. Services offered by link layer include link access, reliable delivery, error detection, error correction, and flow control. A key protocol of link access called *medium access control* addresses the **multiple access problem**: how multiple nodes that share a single broadcast link can coordinate their transmissions so as to avoid collisions. Examples of link layer protocols include Ethernet, the main wired local area network technology (LAN), and 802.11 wireless LAN, and token ring.

Physical Layer: The physical layer is responsible for moving individual bits of data from one node to the next and its protocols are dependent on the actual transmission medium of the link (copper wire, fiber-optic cables etc.)

Overview

In Section 3, we discuss routing algorithms which are a key part of the network layer's functionality. In Section 4, we discuss the basics of queuing theory which is needed for modeling network delay and understanding the performance of queuing strategies. We consider the traditional stochastic queuing theory as well as the adversarial queuing theory. We will then discuss theory of contention resolution protocols in Section 5 which addresses the multiple access problem, a problem handled by the link layer. In Section 6, we will address theoretical issues behind congestion/flow control and resource allocation, an important functionality of the transport layer, in particular, the TCP protocol. In Section 7, we will discuss peer-to-peer networks, the new emerging paradigm at the application layer.

An underlying theme of this chapter is the emphasis on *distributed* or *decentralized* algorithms and protocols. These require only local information (as opposed to global information) which is typically the only information available to the nodes to begin with. Also distributed algorithms are more robust since they don't rely on a central node that might fail. Distributed algorithms can react rapidly to a local change at the point of change. This is especially very useful for problems such as routing. Distributed algorithms are inherently scalable and this is crucial for deployment in a large-scale communication network. The next section gives a brief introduction to distributed computing model and complexity measures that will be used later.

2 Distributed Computing Model

We will focus on the message-passing model of distributed computing where the network is modeled as a graph with nodes (vertices) connected by communication links (edges). There can be weights associated with the links that might denote the delay, or the capacity (bandwidth) of the link. Each edge e supports message passing between the two nodes associated with the endpoints of e . Messages can be sent in both directions (we will assume that the graph is undirected unless otherwise stated). For example, at the network layer of the Internet, nodes correspond to hosts and packet switches (routers), each identified by a unique identifier called the *IP address*, and the messages exchanged between them are basic data transmission units

called *packets*. Alternatively, at the application layer (as in a peer-to-peer network), nodes could correspond to hosts (computers) and edges to TCP/IP connections that are maintained between pairs of nodes.

In the message passing model, information is communicated in the network by exchanging messages. We assume that each node has a unique identity number (e.g., IP address) and at the beginning of computation, each vertex v accepts as input its own identity number and the weights of the edges adjacent to v . Thus, a node has only *local* knowledge limited to itself and its neighbors. We assume that each processor knows its neighbors in the network, and that it communicates directly only with its neighbors.

Two important models can be distinguished based on processor synchronization. In a synchronous model, each processor has an internal clock and the clocks of all processors are synchronized. The processor speeds are uniform and each processor takes the same amount of time to perform the same operation. Thus communication is *synchronous* and occurs in discrete clock “ticks” (time steps). In one time step, each node v can send an arbitrary message of size $O(\log n)$ (n is the number of nodes in the network) through each edge $e = (v, u)$ that is adjacent to v , and the message arrives at u by the end of this time step. (Note that a $O(\log n)$ -size address is needed to uniquely address all nodes.) We will assume that the weights of the edges are at most polynomial in the number of vertices n , and therefore, the weight of a single edge can be communicated in one time step. This model of the distributed computation is called the (synchronous) *CONGEST* ($\log n$) model or simply the *CONGEST* model [40]. The *CONGEST* model is not very realistic for the Internet. However, it has been widely used model to study distributed algorithms and captures the notion that there is a bound on the amount of messages that can be sent in a unit time. The other extreme, is the *LOCAL* model [40] where there is no such bound. We will adopt the *CONGEST* model here.

In an asynchronous model, no assumptions are made about any internal clocks or on the speeds of the processors. The steps in an asynchronous algorithm are determined by conditions or *events* and not by clock ticks. However, we do make two reasonable timing assumptions. First, we assume that messages do arrive (eventually) and in the same order they are sent (i.e., there is FIFO queuing). Second, we assume that if a processor has an event that requires it to perform a task, then it will eventually perform the task. Between the two extreme models, we can define “intermediate” models that are *partially synchronous*, where the processors have some partial information about timing (e.g., almost synchronized clocks or approximate bounds on message delivery time etc.), not the complete information as they do in the synchronous model. Although intermediate models can provide a more realistic model of real networks such as the Internet, we will restrict our attention to synchronous and asynchronous models in this chapter. Algorithms designed

for the synchronous model, can often be translated to work for the asynchronous model (see below), and algorithms for the latter will work for an intermediate model as well.

There are two important complexity measures for comparing distributed algorithms. The first is the time complexity, or the time needed for the distributed algorithm to converge to the solution. In the synchronous model, time is measured by the number of clock ticks called *rounds* (processors compute in “lock step”). For an asynchronous algorithm, this definition is meaningless, since a single message from a node to a neighbor can take a long time to arrive. Therefore the following definition is used: Time complexity in an asynchronous model is the time units from start to finish, assuming that each message incurs a delay of *at most one time unit* [40]. Note that this definition is used only for performance evaluation and has no effect on correctness issues. The second important measure is message complexity, which measures the total number of messages (each of size $O(\log n)$) that are sent between all pairs of nodes during the computation.

We will assume synchronous (*CONGEST*) model unless otherwise stated, since it is simpler and easier to design algorithms for this model. Using a tool called *synchronizers* one can transform a synchronous algorithm to work in an asynchronous model with no increase in time complexity and at the cost of some increase in the message complexity [40].

3 Routing Algorithms

We discuss three basic routing modes that are used in the Internet: unicast, broadcast, and multicast. In unicast, a node sends a packet to another specific node; in broadcast, a node send a packet to every node in the network; in multicast, a node sends a packet to a subset of nodes. We focus on fundamental distributed network algorithms that arise in these routing modes.

3.1 Unicast Routing

Unicast routing (or just simply routing) is the process of determining a “good” path or route to send data from the source to the destination. Typically, a good path is one that has the least cost. Consider a weighted network $G = (V, E, c)$ with positive real-valued edge (link) costs given by the cost function c . The cost of a path $p = (e_1, \dots, e_k)$ is defined as $c(p) = \sum_{i=1}^k c(e_i)$. For a source-destination pair $(s, t) \in (V \times V)$, the goal is to find a **least-cost (or shortest) path**, i.e., a path from s to t in G that has minimum cost. If all edges in the graph have cost 1, the shortest path is the path with the smallest number of links between the

source and destination. The Internet uses least-cost path routing. One way to classify routing algorithms is according to whether they are global or local:

- A global routing algorithm computes the least-cost path between a source and a destination using complete, global knowledge of the network. In practice, it happens to be referred to as a link state (LS) algorithm.
- A local routing algorithm computes the least-cost path in an iterative, distributed fashion. No node has complete information about all the edge costs. In practice, it happens to be referred to as a **distance vector** (DV) algorithm.

The link state and distance vector algorithms are essentially the *only* routing algorithms used in the Internet today.

3.1.1 A Link State Routing Algorithm

A link state (LS) algorithm knows the global network topology and all link costs. One way to accomplish this is by having each node broadcast its identity number and costs of its incident edges to all other nodes in the network using a broadcasting algorithm, e.g., flooding. (Broadcast algorithms are described in Section 3.3.) Each node can then run the (centralized) link state algorithm and compute the same set of shortest paths as any other node. A well-known LS algorithm is the *Dijkstra's algorithm* for computing least-cost paths. This algorithm takes as input a weighted graph $G = (V, E, c)$, a source vertex $s \in V$ and computes shortest paths (and their values) from s to all nodes in V . Dijkstra's algorithm and its run-time analysis is given in Chapter 6. Note that this run-time is the time needed to run the algorithm in a single node. The message complexity and time complexity of the algorithm is determined by the broadcast algorithm. For example, if broadcast is done by flooding (cf. Section 3.3.1) then the message complexity is $O(|E|^2)$, since $O(|E|)$ messages have to be broadcast, each of which causes $O(|E|)$ messages to be sent by flooding. The time complexity is $O(|E|D)$, where D is the **diameter** of the network. Internet's Open Shortest Path First (OSPF) protocol uses a LS routing algorithm as mentioned above. Since the LS algorithm is centralized, it may not scale well when the networks become larger.

3.1.2 A Distance Vector Algorithm

The distance vector (DV) algorithm is distributed and asynchronous. It is distributed because information is exchanged only between neighbors. Nodes then perform local computation and distribute the results back to its neighbors. This process continues till no more information is exchanged between neighbors. The algorithm is asynchronous in that it does not require all of the nodes to operate in lock step with each other.

The DV algorithm that is described below is called the distributed *Bellman-Ford* (DBF) algorithm. It is used in the Routing Information Protocol (RIP) and the Border Gateway Protocol (BGP) of the Internet.

We will describe the basic idea behind the DBF algorithm ([15, 2]). Suppose we want to compute the shortest (least-cost) path between s and all other nodes in a given undirected graph $G = (V, E, c)$ with real-valued positive edge weights. During the algorithm each node x maintains a distance label $a(x)$ which is the current known shortest distance from s to x , and a variable $p(x)$ which contains the identity of the previous node on the current known shortest path from s to x . Initially, $a(s) = 0$, $a(x) = \infty$, and $p(x)$ is undefined for all $x \neq s$. When the algorithm terminates, $a(x) = d(s, x)$, where $d(s, x)$ is the shortest path distance between s and x , and $p(x)$ holds the neighbor of x on the shortest path from x to s . Thus, if node x wants to route to s along the shortest path it has to forward its data to $p(x)$.

The DBF consists of two basic rules: the update rule and the send rule. The update rule determines how to update the current label according to a message from a neighboring node. The send rule determines what values to send to its neighbors and is applied whenever a node adopts a new label.

Update rule: Suppose x with a label $a(x)$ receives $a(z)$ from a neighbor z . If $a(z) + c(z, x) < a(x)$, then it updates $a(x)$ to $a(z) + c(z, x)$ and sets $p(x)$ to be z . Otherwise $a(x)$ and $p(x)$ are not changed.

Send rule: Let $a(x)$ be a new label adopted by x . Then x sends $a(x)$ to all its neighbors.

Correctness and Analysis of the DBF algorithm

We assume a *synchronous* setting where computation takes place in discrete *rounds*, i.e., all nodes simultaneously receive messages from their neighbors, perform the update rule (if necessary), and send the message to their neighbors (if update was performed). The following theorem gives the correctness and complexity of the DBF algorithm.

Theorem 3.1 *If all nodes work in a synchronous way, then the DBF algorithm terminates after at most n rounds. When it terminates, $a(x) = d(s, x)$ for all nodes x . The message complexity is $O(n|E|)$.*

Proof: Fix a vertex $x \in V$, we prove that the algorithm computes a shortest path from s to x .

Let $P = v_0, v_1, \dots, v_k$, where $v_0 = s$ and $v_k = x$ be a shortest path from s to x . Note that $k < n$.

We prove by induction on i that after the i th round, the algorithm has computed the shortest path from s to v_i , i.e., $a[v_i] = d(s, v_i)$.

The hypothesis holds for $v_0 = s$ in round zero.

Assume that it holds for $j \leq i - 1$. After the i th iteration,

$$a[v_i] \leq a[v_{i-1}] + c(v_{i-1}, v_i) \quad (1)$$

which is the shortest path from s to v_i , since P is a shortest path from s to v_i , and the right hand side is the distance between s to v_i on that path.

Since, in each round $O(|E|)$ messages are exchanges, the total message complexity is $O(n|E|)$. \square

From the above proof, it can be seen that the algorithm will compute the correct values even if the nodes operate asynchronously. An important observation is that the algorithm is “self-terminating” — there is no signal that the computation should stop; it just stops. The above rules can easily be generalized to compute the shortest path between *all pairs* of nodes, by maintaining in each node x a distance label $a_y(x)$ for every $y \in V$. This is called the **distance vector** of x . Each node stores its own distance vector and the distance vectors of each of its neighbors. Whenever something changes, say the weight of any of its incident edges or its distance vector, the node will send its distance vector to all of its neighbors. The receiving nodes then update their own distance vectors according to the update rule.

A drawback of the DBF algorithm is that convergence time can be made arbitrarily large if the initial distance labels are not correct. Consider the following simple network consisting of 4 nodes and 3 links shown in Figure 1. The goal is to compute the shortest paths to D . Initially each link has weight 1 and each node had calculated the shortest path distance to D . Thus the distances of A , B , and C to D are 3, 2, and 1 respectively. Now suppose the weight of edge (C, D) changes from 1 to a large positive number, say L . Assuming a synchronous behavior, in the subsequent iteration, C will set its distance label to D as 3, since B supposedly has a path to D of length 2. In the following iteration B will change its distance estimate to 4, since the best path it has is through C . This will continue until C distance label reaches L . Thus the number of iterations taken to converge to the correct shortest path is proportional to L . This problem, is referred to as the “**count-to-infinity**” **problem** and shows that the algorithm can be very slow in reacting to a change in an edge cost. Some heuristics have been proposed to alleviate this problem; we refer to [5, 27] for details.

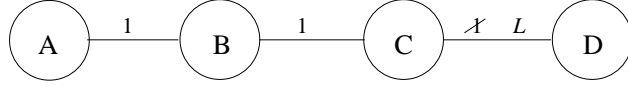


Figure 1: An example network for count-to-infinity problem.

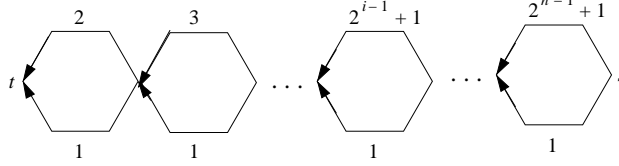


Figure 2: An example where the number of messages in the DBF algorithm is $\Omega(2^n)$.

An Approximate Distributed Bellman-Ford Algorithm

The DBF algorithm can suffer from exponential message complexity in an asynchronous setting. For example, consider the graph shown in Figure 2. There are 2^n distinct paths from s to t , each one with a distinct length. It is possible, in an asynchronous environment, to create an execution instance of DBF such that t will receive $\Omega(2^n)$ messages [2]. Awerbuch et al. [2] proposed a simple modification to the DBF algorithm that will result in a polynomial message complexity. However, the modified algorithm may not compute a shortest path, but will instead compute an *approximate* shortest path, in particular, the paths computed can be worse than the shortest path by *at most* a constant factor. Thus this can be called as a *distributed approximation algorithm*. The only difference between the modified algorithm and the DBF algorithm is in the update rule:

Multiplicative update rule: Let $\alpha = 1 + 1/n$. Suppose x , with a label $a(x)$ receives $a(z)$ from a neighbor z . If $a(z) + c(x, z) < a(x)/\alpha$, then $a(x)$ is updated to $a(z) + c(x, z)$ and $p(x)$ is set to z .

The following theorem gives the performance of the modified DBF. For a proof we refer to [2].

Theorem 3.2 *The length of the computed path between s and x (for any node x) at the end of the execution of the algorithm is at most $e \cdot d(s, x)$, where e is the base of the natural logarithm. The number of messages sent is bounded by $O(|E|n \log(n\Delta))$, where Δ is the largest edge cost.*

3.2 Multicommodity flow-based Routing

A drawback of shortest path routing is that each source-destination is selected independent of other paths. It is quite possible that least-cost paths can cause a high congestion, i.e., many paths may go through the same edge. Alternate to least-cost paths routing, a broad class of routing algorithms is based on viewing

packet traffic as flows between sources and destinations in a network. In this approach the routing problem is formulated as a constrained optimization problem known as a *multicommodity network flow* problem with the goal of optimizing some appropriate global objective. In a multicommodity flow problem, we are given a directed network $G = (V, E)$ with capacity $C(i, j)$ for each directed edge (i, j) . Let $r_i(j)$ be the traffic rate (or *demand*) from source i to destination j . While routing traffic from source to destination, we obtain a *flow*, i.e., traffic can be split and routed along multiple paths. Let $f_{ik}(j)$ denote the traffic flow destined for j being sent on edge (i, k) . The goal is to find paths between source-destination pairs and to determine the amount of flow on these paths under a set of constraints and objectives. Two types of constraints are standard:

Capacity constraint: The flow through an edge cannot exceed its capacity, i.e.,

$$\sum_j f_{ik}(j) \leq C(i, k). \quad (2)$$

Flow conservation constraint: The amount of flow for j coming into node i equals the amount of traffic for j leaving node i , i.e.,

$$r_i(j) + \sum_k f_{ki}(j) = \sum_k f_{ik}(j). \quad (3)$$

We have the following different problems based on different objective functions:

Maximum multicommodity flow: The objective is to maximize the sum of the traffic rates, i.e., total throughput, that can be delivered in the network. Formally, we want to

$$\text{maximize } \sum_{i,j} r_i(j),$$

subject to capacity and flow conservation constraints.

Maximum concurrent flow problem: The objective is to maximize the *fraction of traffic* routed for each source-destination pair. This can be considered a “fairer” objective compared to the maximum multicommodity flow. Formally, the objective is to maximize the fraction μ subject to capacity and flow constraints.

The latter may be rewritten as:

$$\mu r_i(j) + \sum_k f_{ki}(j) = \sum_k f_{ik}(j). \quad (4)$$

Both the above problems can be solved using linear programming formulations. These algorithms are

complex and are not easy to implement distributively. Awerbuch and Leighton [3] gave a local control algorithm for solving the above problems. It gives an $(1 + \epsilon)$ -approximation for a desired $\epsilon > 0$. See Chapter 7 for a description of this algorithm. This algorithm can be implemented in a distributed manner.

Minimum delay routing problem (MDRP): The objective is to minimize the average delay subject to capacity and flow conservation constraints. The average delay on a link is assumed to be an increasing (convex) function of the amount of traffic on the link. Thus the average delay per packet on link (i, k) is given by $D_{ik}(f_{ik})$ where D_{ik} is a continuous and increasing (convex) function and f_{ik} is the total traffic on link (i, k) . ($D_{ik}(f_{ik})$ tends to infinity as f_{ik} approaches C_{ik} .) The objective is to minimize the *total* average delay given by:

$$D_T = \sum_{i,k} D_{ik}(f_{ik}). \quad (5)$$

A Distributed Algorithm for Minimum Delay Routing

The problem is complicated since the objective function is not linear and the challenge is to come up with a distributed algorithm. We briefly describe a distributed algorithm due to Gallager [17].

Let $t_i(j)$ be the total flow (traffic) at node i destined for j ; thus $t_i(j) = r_i(j) + \sum_k f_{ki}(j)$.

Let $\phi_{ik}(j)$ be the fraction of the node flow $t_i(j)$ that is routed over edge (i, k) . Since node flow $t_i(j)$ at node i is the sum of the input traffic and the traffic routed to i from other nodes,

$$t_i(j) = r_i(j) + \sum_{\ell} t_{\ell}(j)\phi_{\ell i}(j) \quad \forall i, j. \quad (6)$$

The above equation implicitly expresses the flow conservation at each node. Now we can express f_{ik} , the traffic on link (i, k) as:

$$f_{ik} = \sum_j t_i(j)\phi_{ik}(j). \quad (7)$$

The variable set ϕ is called as the *routing variable* and satisfies the following conditions:

1. $\phi_{ik}(j) = 0$, if (i, k) is not an edge or if $i = j$.
2. $\sum_k \phi_{ik}(j) = 1$ for all j .

Note that the traffic flow set $t = \{t_i(j)\}$ and link flow set $F = \{f_{ik}\}$ can be obtained from $r = \{r_i(j)\}$ and $\phi = \{\phi_{ik}(j)\}$. Therefore D_T can be expressed as a function of r and ϕ using equations 6 and 7. The MDRP can be restated as follows:

For a given network $G = (V, E)$, and input traffic flow set $r_i(j)$, and delay function $D_{ik}(f_{ik})$ for each

link (i, k) , find the variable set ϕ that minimizes the average delay subject to the above conditions on ϕ and the capacity and flow constraints.

We will assume that for each i, j ($i \neq j$) there is a routing path from i to j , thus there is a sequence of nodes i, k, ℓ, \dots, m , such that $\phi_{ik}(j) > 0, \phi_{k\ell}(j) > 0, \dots, \phi_{mj}(j) > 0$. Gallager shows that the above conditions guarantee that the set of Equations 6 has a unique solution for t .

Necessary and sufficient conditions for optimality

Gallager derived the necessary and sufficient conditions that must be satisfied to solve MDRP. These conditions are summarized in the theorem stated below.

Theorem 3.3 [17] *The necessary condition for a minimum of D_T with respect to ϕ for all $i \neq j$ and $(i, k) \in E$ is*

$$\frac{\partial D_T}{\partial \phi_{ik}(j)} \begin{cases} = \lambda_{ij} & : \phi_{ik}(j) > 0 \\ \geq \lambda_{ij} & : \phi_{ik}(j) = 0 \end{cases} \quad (8)$$

where λ_{ij} is some positive number.

The sufficient condition to minimize D_T with respect to ϕ is for all $i \neq j$ and $(i, k) \in E$ is

$$D'_{ik}(f_{ik}) + \frac{\partial D_T}{\partial r_k(j)} \begin{cases} = \frac{\partial D_T}{\partial r_i(j)} & : \phi_{ik}(j) > 0 \\ \geq \frac{\partial D_T}{\partial r_k(j)} & : \phi_{ik}(j) = 0 \end{cases} \quad (9)$$

$\frac{\partial D_T}{\partial r_k(j)}$ is called the marginal distance from i to j and $\frac{\partial D_T}{\partial \phi_{ik}(j)}$ is called the marginal delay and $D'_{ik}(f_{ik})$ is the marginal link delay.

Thus Gallager's optimality conditions mandates two properties:

(1) For a given node i and a destination j , the marginal link delays should be *same* for all links (i, k) for which there is non-zero flow i.e., $\phi_{ik}(j) > 0$. Furthermore, this marginal delay must be less than or equal to the marginal delay on the links on which the traffic flow is zero. This is the necessary condition.

(2) Under optimal routing at node i with respect to a particular destination j , the marginal distance through a neighbor k plus the marginal link delay should be equal to the marginal distance from node i if traffic is forwarded through k . It should be greater if no traffic is forwarded through k . This is the sufficiency condition.

Gallager's algorithm for MDRP: Gallager's algorithm iteratively computes the routing variables $\phi_{ik}(j)$ in

a distributed manner. The idea is to progress incrementally towards the optimality conditions. Each node i incrementally decreases those routing variables $\phi_{ik}(j)$ for which the marginal delay $D'_{ik}(f_{ik}) + \frac{\partial D_T}{\partial r_k(j)}$ is large and increases those for which it is small. The decrease is by a small quantity ϵ ; the excess fraction is moved over to the link (i, m) that has the smallest marginal delay (the current best link).

Gallager proves that for suitably small ϵ , the above iterative algorithm converges to the minimum delay routing solution. The choice of ϵ is critical in determining the convergence and its rate. A small ϵ may take a large time to converge, while a large ϵ may cause the system to diverge or oscillate around the minimum. Gallager's algorithm uses a blocking technique to make sure that the routes are *loop-free* at every instant. Loop-freedom ensures that the traffic that is routed along a path does not come back along a cycle. The algorithm ensures that if the routing variables are loop-free to begin with, then they will be loop-free during the execution of the algorithm. Thus, one way to start the algorithm is to assign the routing variables based on shortest path routing, i.e., they are set to 1 for the edges along the shortest paths, and 0 otherwise. To change the routing variables, each node needs to calculate marginal delays. This can be done in a distributed manner by propagating the value from the destination (its marginal delay is zero) as follows. For each destination j , each node i waits until it has received the value $\frac{\partial D_T}{\partial r_k(j)}$ from each of its *downstream* neighbors $k \neq j$ (i.e., neighbors that are closer to j on a routing path from i to j). The node i then calculates $\frac{\partial D_T}{\partial r_i(j)}$ (additional information needed are $\phi_{ik}(j)$ and $D'_{ik}(f_{ik})$ which are known locally) and broadcasts to all of its neighbors. Loop-freedom is essential to guarantee that this procedure is free from deadlocks [17].

Confluent Flow Routing

Most flows in today's communication networks, especially the Internet, are **confluent**. A flow is said to be confluent if all the flows arriving at a node destined for a particular destination departs from the node along a single edge. Internet flows are confluent because Internet routing is destination-based. (Recall from our discussion of shortest-path routing, the forwarding tables of the routers are set to have one entry per destination.) Destination-based routing makes forwarding simple and the forwarding table size linear in the number of the nodes in the network. Both shortest path routing and BGP routing is destination based. One can study multicommodity flow problems with the condition that flows must be confluent. The primary goal is to find the confluent flows that minimize maximum congestion in the network. The **congestion** of an edge (node) is defined as the total amount of flow going through the edge (node). Formally, we consider the following problem studied in [11].

Minimum congestion ratio routing: We are given a directed graph $G = (V, E)$ with n nodes, m edges and

capacities $C(i, j)$ for each $(i, j) \in E$. Represent the traffic associated with a particular destination as a separate commodity. Let there be a total of k commodities. Let commodity i be associated with destination t_i and a set $S_i \subset V$ sources. The commodity demand, given by a function $d : \{1, \dots, k\} \times V \rightarrow \mathcal{R}^+$, specifies the demand of each commodity for each vertex. A flow $f : \{1, \dots, k\} \times V \times V \rightarrow \mathcal{R}$ (specifies the amount of flow of each commodity type through each pair of vertices) is confluent if for any commodity i , there is at most one outgoing flow at any node v . (It is easy to see that for any commodity i , flow f induces a set of **arborescences**, each of which is rooted at a destination t_i .) Given a flow f , the **congestion ratio** at an edge (u, v) with $c(u, v) > 0$, denoted by $r(u, v)$ is the ratio between the total flow $\sum_{i \in \{1, \dots, k\}} f(i, u, v)$ on this edge and the capacity of the edge $C(u, v)$. The congestion ratio of flow f is the maximum congestion ratio among all edges. The minimum congestion ratio problem is to find a confluent flow to satisfy all the demands with minimum congestion ratio.

The above problem is NP-hard to solve optimally [12]. Hence the focus is to look for an approximately optimal solution that can be implemented in a distributed fashion. We present an algorithm due to Chen et al. [11] called as the *locally independent rounding algorithm (LIRA)*. The main idea is to cast the above problem as an integer linear program and then solve its relaxation. The relaxation is the standard multicommodity flow problem where confluence constraints need not be satisfied. Let $D_i = \sum_{v \in V} d(i, v)$, i.e., the total demand for commodity i . Let $x : \{1, \dots, k\} \times V \times V \rightarrow [0, 1]$ and $\rho \in \mathcal{R}^+$. $x(i, u, v)$ will denote the fraction of the total commodity i that enters node u that leaves for v . If the flow is confluent $x(i, u, v)$ will be 0 or 1 for all $i \in \{1, \dots, k\}$ and $u, v \in V$. If $x(i, u, v)$ is allowed to be fractional, we get a **splittable** flow (need not be confluent). The following linear program computes a splittable flow with minimum congestion

ratio.

Minimize r

subject to

$$\sum_{v \in V} f(i, u, v) = d(i, u), \quad \forall u \neq t_i, \forall i \quad (10)$$

$$\sum_{i \in \{1, \dots, k\}} f(i, u, v) \leq rc(u, v), \quad \forall u, v \in V \quad (11)$$

$$0 \leq x(i, u, v) \leq 1, \forall u, v \in V, \quad \forall i \in \{1, \dots, k\} \quad (12)$$

$$\sum_{w \in V} x(i, v, w) = 1, \forall v \in V, \quad \forall i \in \{1, \dots, k\} \quad (13)$$

$$0 \leq f(i, u, v) \leq D_i x(i, u, v) \forall u, v \in V, \quad \forall i \in \{1, \dots, k\} \quad (14)$$

Equation 10 captures flow conservation and Inequality 11 captures capacity constraints. The above multicommodity problem can be solved approximately in a distributed manner by using the local control algorithm of Awerbuch and Leighton [3] (see Section 3.2). This gives a $1 + \epsilon$ approximation. The splittable flow computed by this algorithm is then made confluent by using the technique of *randomized rounding* (see Chapter 34) as follows. Let f be the flow computed by the algorithm of [3]. Then each node chooses for each commodity a unique outgoing edge independently at random. Node u chooses, for commodity i , edge (u, v) with probability

$$p(i, u, v) = \frac{f(i, u, v)}{\sum_{(u, v') \in E} f(i, u, v')}. \quad (15)$$

This rounding algorithm can be easily implemented in a distributed manner, since each node makes its own choice based on local information.

The following theorem can be shown [11].

Theorem 3.4 *Given a splittable flow f with congestion ratio C , the above rounding algorithm produces a confluent flow ϕ with $O(\max(C, D/c_{\min} \log n))$ congestion ratio with high probability, where $D = \max_i D_i$ and c_{\min} is the minimum edge capacity.*

Note that if $C = \Omega(D/c_{\min} \log n)$, then LIRA is a constant factor approximation algorithm for the multicommodity flow problem. If $C = \Omega(D/c_{\min})$ then LIRA achieves a logarithmic approximation. These performance guarantees do not hold if these conditions are not true. Note that $\Omega(D/c_{\max} \delta)$ is a lower bound on C where δ is the maximum degree of a node. If capacities are more or less uniform and the

maximum degree is not large then these conditions are reasonable. Stronger bounds on various special cases of the confluent flow problem have been established. We refer to [12, 10] for details.

3.3 Broadcast Routing

Broadcasting is another important communication mode: sending a message from a source node to all other nodes of the network. We will consider two basic broadcasting approaches: flooding and spanning tree-based routing.

3.3.1 Flooding

Flooding is a natural and basic algorithm for broadcast. Suppose a source node s wants to send a message to all nodes in the network. s simply forwards the message over all its edges. Any vertex $v \neq s$, upon receiving the message for the *first* time (over an edge e) forwards it on every other edge. Upon receiving the message again it does nothing. It is easy to check that this yields a correct broadcast algorithm. The complexity of flooding can be summarized by:

Theorem 3.5 *The message complexity of flooding is $\Theta(|E|)$ and the time complexity is $\Theta(\text{Diam}(G))$ in both the synchronous and asynchronous models.*

Proof: The message complexity follows from the fact that each edge delivers the message at least once and at most twice (one in each direction). To show the time complexity, we use induction on t to show that after t time units, the message has already reached every vertex at a distance of t or less from the source.

□

3.3.2 Minimum Spanning Tree (MST) Algorithms

The **Minimum Spanning Tree** (MST) problem is an important and commonly occurring primitive in the design and operation of communication networks. The formal definition of the problem, some properties of MST, and several sequential algorithms are given in Chapter 6. Of particular interest here, is that an MST can be used naturally for broadcast. Any node that wishes to broadcast simply sends messages along the spanning tree ([27]). The advantage of this method over flooding is that redundant messages are avoided. The message complexity is $O(n)$ which is optimal.

Here we focus on distributed algorithms for this problem and few more properties of MST. The first distributed algorithm for the MST problem was given by Gallager, Humblet, and Spira [18] in 1983. This algorithm is known as GHS algorithm and will work in an asynchronous model also.

GHS Algorithm

GHS algorithm assumes that the edge weights are distinct. If all the edges of a connected graph have distinct weights, then the MST is unique. Suppose, to the contrary, that there are two different MSTs, T and T' . Let e be the minimum-weight edge that is in T but not in T' . The graph $\{e\} \cup T'$ must contain a cycle, and at least one edge in this cycle, say e' , is not in T , as T contains no cycles. Since the edge weights are all distinct and e' is in one but not both of the trees, weight of e is strictly less than the weight of e' . Thus $\{e\} \cup T' - \{e'\}$ is a spanning tree of smaller weight than T' ; this is a contradiction.

We are given an undirected graph $G = (V, E)$. Let T be the MST on G . A *fragment* F of T is defined as a connected subgraph of T , that is, F is a subtree of T . An *outgoing edge* of a fragment is an edge in E where one adjacent node to the edge is in the fragment and the other is not. The *minimum-weight outgoing edge (MOE)* of a fragment F is the edge with minimum weight among all outgoing edges of F . As an immediate consequence of the blue rule for MST (see Chapter 6 in [48]), the MOE of a fragment $F = (V_F, E_F)$ is an edge of the MST. Consider a cut $(V_F, V - V_F)$ of G . The MOE of F is the minimum-weight edge in the cut $(V_F, V - V_F)$, and therefore the MOE is an edge of the MST. Thus adding the MOE of F to F along with the node at the other end of MOE yields another fragment of the MST. The algorithm starts with each individual node as a fragment by itself and ends with one fragment — the MST. That is, at the beginning, there are $|V|$ fragments, and at the end, a single fragment which is the MST. All fragments find their MOE simultaneously in parallel. Initially, each node (a singleton fragment) is a core node; subsequently each fragment will have one core node (determined as explained below). To find the MOE of a fragment, the core node in the fragment broadcasts a message to all nodes in the fragment using the edges in the fragment. Each node in the fragment, after receiving the message, finds the minimum outgoing edge adjacent to it and reports to the core node. Once the MOE of the fragment is found, the fragment attempts to combine with the fragment at the other end of the edge. Each fragment has a level. A fragment containing only a single node is at level 0. Let the fragment F be at level L , the edge e be the MOE of F , and the fragment F' be at the other end of e . Let L' be the level of F' . We have the following rules for combining fragments:

- (1) If $L < L'$, fragment F and F' are combined into a new fragment at level L' , and the core node of F' is the core node of the new fragment.

(2) If $L = L'$ and fragments F and F' have the same minimum-weight outgoing edge, F and F' are combined into a new fragment at level $L + 1$. The node incident on the combining edge with the higher identity number is the core node of the new fragment.

(3) Otherwise, fragment F waits until fragment F' reaches a level high enough to apply any of the above two rules of combining.

The waiting of the fragments in the above procedure cannot cause a deadlock. The waiting is done to reduce the communication cost (number of messages) required for a fragment to find its MOE. The communication cost is proportional to the fragment size, and thus communication is reduced by small fragments joining into large ones rather than vice versa. The maximum level a fragment can reach is $\lg n$, where $n = |V|$. The algorithm takes $O(n \lg n + |E|)$ messages and $O(n \lg n)$ time. It can be shown that any distributed algorithm for MST problem requires $\Omega(n \lg n + |E|)$ messages [49]. Thus the communication (message) complexity of GHS algorithm is optimal. However, its time complexity is not optimal.

Kutten and Peleg's Algorithm

Kutten and Peleg's [28] distributed MST algorithm runs in $O(D + \sqrt{n} \log^* n)$ time, where D is the diameter of the graph G . The algorithm consists of two parts. In the first part, similar to GHS algorithm, this algorithm begins with each node as a singleton fragment. Initially, all of these singleton fragments are *active* fragments. Each active fragment finds its MOE and merges with another fragment by adding MOE to the MST. Once the depth of a fragment reaches \sqrt{n} , it becomes a *terminated* fragment. A terminated fragment stops merging onto other fragments. However, an active fragment can merge onto a terminated fragment. The depth of a fragment is measured by broadcasting a message from the root of the fragment up to the depth of at most \sqrt{n} and by using a counter in the message. At the end of the first part, the size (number of nodes) of a fragment is at least \sqrt{n} , and there are at most \sqrt{n} fragments. The first part of the algorithm takes $O(\sqrt{n} \log^* n)$ time (see [28] for details).

In the second part of the algorithm, a breadth-first tree B is built on G . Then the following pipeline algorithm is executed. Let $r(B)$ be the root of B . Using the edges in B , $r(B)$ collects weights of the inter-fragment edges, computes the minimum spanning tree, T' , of the fragments by considering each fragment as a super node. It then broadcasts the edges in T' to the other nodes using the breadth-first tree B .

This pipeline algorithm follows the principle used by Kruskal's algorithm. Each node v , except the root r , maintains two lists of inter-fragment edges, Q and U . Initially, Q contains only the inter-fragment edges adjacent to v , and U is empty. At each pulse, v sends the minimum-weight edge in Q that does not create a

cycle with the edges in U to its parent and moves this edge from Q to U . If Q is empty, v sends a terminate message to its parent. The parent after receiving an edge from a child, adds the edge in its Q list. A leaf node starts sending edges upwards at pulse 0. An intermediate node starts sending at the first pulse after it has received at least one message from each of its children.

Observe that the edges reported by each node to its parent in the tree are sent in nondecreasing weight order, and each node sends at most \sqrt{n} edges upward to its parent. Using the basic principle of Kruskal's algorithm, it can be easily shown that the root $r(B)$ receives all the inter-fragment edges required to compute T' correctly. To build B , it takes $O(D)$ time. Since the depth of B is D and each node sends at most \sqrt{n} edges upward, the pipeline algorithm takes $O(D + \sqrt{n})$ time. Thus the time complexity of Kutten and Peleg's algorithm is $O(D + \sqrt{n} \log^* n)$. The communication complexity of this algorithm is $O(|E| + n^{1.5})$.

Peleg and Rabinovich [41] showed that $\tilde{\Omega}(D + \sqrt{n})$ time¹ is required for distributed construction of MST. Elkin [16] showed that even finding an approximate MST in distributed time on graphs of small diameter (e.g., $O(\log n)$) within a ratio H requires $\Omega(\sqrt{\frac{n}{H \log n}})$ time.

Khan and Pandurangan's Approximation Algorithm

Khan and Pandurangan [24] gave an algorithm for $O(\log n)$ -approximate MST that runs in $\tilde{O}(D + L)$ time. L is called local shortest path diameter. L can be small in many classes of graphs, especially those of practical interest such as wireless networks. In the worst case, L can be as large as $n - 1$.

Khan and Pandurangan's algorithm ([24]) is based on simple scheme called the Nearest Neighbor Tree (NNT) scheme. The tree constructed using this scheme is called *nearest neighbor tree*, which is an $O(\log n)$ -approximation to MST. The *NNT scheme* is as follows: (1) each node chooses a unique *rank* from a totally ordered set, and (2) each node, except the one with the highest rank, connects via the *shortest path* to the *nearest* node of higher rank, that is, add the edges in the shortest path to the NNT. The added edges can create cycle with the existing (previously-added) edges of NNT. Cycle formation is avoided by removing some existing edges.

The following notations and definitions are used to describe the algorithm.

$|Q(u, v)|$ or simply $|Q|$ — the number of edges in path Q from u to v .

$w(Q)$ — the weight of the path Q , which is defined as the sum of the weights of the edges in path Q .

$P(u, v)$ — a shortest path (in the weighted sense) from u to v .

¹The $\tilde{\Omega}$ notation hides logarithmic factors.

$d(u, v)$ — the (weighted) distance between u and v , that is, $d(u, v) = w(P(u, v))$.

$W(v)$ — the weight of the largest edge adjacent to v . $W(v) = \max_{(v,x) \in E} w(v, x)$.

$l(u, v)$ — the number of the edges in the shortest path from u to v . If there are more than one shortest path from u to v , $l(u, v)$ is the number of edges of the shortest path having the least number of edges, i.e., $l(u, v) = \min\{|P(u, v)| \mid P(u, v) \text{ is a shortest path from } u \text{ to } v\}$.

ρ -neighborhood. ρ -neighborhood of a node v , denoted by $\Gamma_\rho(v)$, is the set of the nodes that are within distance ρ from v . $\Gamma_\rho(v) = \{u \mid d(u, v) \leq \rho\}$.

(ρ, λ) -neighborhood. (ρ, λ) -neighborhood of a node v , denoted by $\Gamma_{\rho, \lambda}(v)$, is the set of all nodes u such that there is a path $Q(v, u)$ such that $w(Q) \leq \rho$ and $|Q| \leq \lambda$. Clearly, $\Gamma_{\rho, \lambda}(v) \subseteq \Gamma_\rho(v)$.

Local Shortest Path Diameter (LSPD). LSPD is denoted by $L(G, w)$ (or L for short) and defined by $L = \max_{v \in V} L(v)$, where $L(v) = \max_{u \in \Gamma_{W(v)}(v)} l(u, v)$.

Rank selection. The nodes select unique ranks as follows. First a leader is elected by a leader election algorithm (e.g., see [35]). (Or assume that there is one such node, say which initiates the algorithm.) Let s be the leader node. The leader picks a number $p(s)$ from the range $[b - 1, b]$, where b is a (real) number arbitrarily chosen by s , and sends this number $p(s)$ along with its ID (identity number) to all of its neighbors. As soon as a node u receives the first message from a neighbor v , it picks a number $p(u)$ from $[p(v) - 1, p(v))$ so that it is smaller than $p(v)$, and sends $p(u)$ and $ID(u)$ to its neighbors. If u receives another message later from another neighbor v' , u simply stores $p(v')$ and $ID(v')$, and does nothing else. $p(u)$ and $ID(u)$ constitute u 's rank $r(u)$ as follows. For any two nodes u and v , $r(u) < r(v)$ iff i) $p(u) < p(v)$, or ii) $p(u) = p(v)$ and $ID(u) < ID(v)$.

At the end of execution of the above procedure of rank selection, (i) each node knows the ranks of all of its neighbors, (ii) the leader s has the highest rank among all nodes in the graph, and (iii) each node v , except the leader, has one neighbor u , i.e. $(u, v) \in E$, such that $r(u) > r(v)$.

Connecting to a higher-ranked node. Each node v , except the leader s , executes the following algorithm simultaneously to find the nearest node of higher rank and connect to it. Each node v needs to explore only the nodes in $\Gamma_{W(v)}(v)$ to find a node of higher rank. This is because at the end of rank selection procedure, v has at least one neighbor u such that $r(u) > r(v)$, and if u is a neighbor of v , then $d(u, v) \leq W(v)$.

Each node v executes the algorithm in *phases*. In the first phase, v sets $\rho = 1$. In the subsequent phases, it doubles the value of ρ ; that is, in the i th phase, $\rho = 2^{i-1}$. In a phase of the algorithm, v explores the nodes

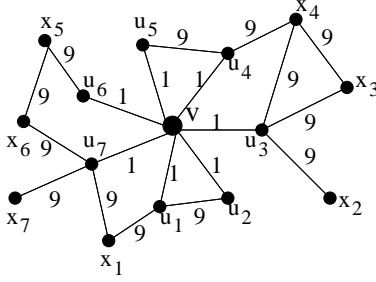


Figure 3: A network with possible congestion in the edges adjacent to v . Weight of the edges (v, u_i) is 1 for all i , and 9 for the rest of the edges. Assume $r(v) < r(u_i)$ for all i .

in $\Gamma_\rho(v)$ to find a node u (if any) such that $r(u) > r(v)$. If such a node with higher rank is not found, v continues to the next phase with ρ doubled. Node v needs to increase ρ to at most $W(v)$. Each phase of the algorithm consists of one or more *rounds*. In the first round, v sets $\lambda = 1$. In the subsequent rounds, values for λ are doubled. In a particular round, v explores all nodes in $\Gamma_{\rho, \lambda}(v)$. At the end of each round, v counts the number of nodes it has explored. If the number of nodes remain the same in two successive rounds of the same phase (that is, v already explored all nodes in $\Gamma_\rho(v)$), v doubles ρ and starts the next phase. If at any point of time v finds a node of higher rank, it terminates its exploration.

Since all of the nodes explore their neighborhoods simultaneously, many nodes may have overlapping ρ -neighborhoods. This might create congestion of the messages in some edges that may result in increased running time of the algorithm, in some cases by a factor of $\Theta(n)$. Consider the network given in Figure 3. If $r(v) < r(u_i)$ for all i , when $\rho \geq 2$ and $\lambda \geq 2$, an exploration message sent to v by any u_i will be forwarded to all other u_i s. Note that values for ρ and λ for all u_i s will not necessarily be the same at a particular time. Thus congestion at any edge (v, u_i) can be as much as the number of such nodes u_i , which can be, in fact, $\Theta(n)$ in some graphs. However, to improve the running time of the algorithm, the congestions on all edges is controlled to be $O(1)$ by sacrificing the quality of the NNT, by a constant factor, as explained below.

If at any time step, a node v receives more than one, say $k > 1$, messages from different originators u_i , $1 \leq i \leq k$, v forwards only one message and replies back to the other originators as follows. Let u_i be in phase ρ_i . For any pair of originators u_i and u_j , if $r(u_i) < r(u_j)$ and $\rho_j \leq \rho_i$, v sends back a *found* message to u_i telling that u_i can connect to u_j (note that the weight of the connecting path $w(Q(u_i, u_j)) \leq 2\rho_i$) instead of forwarding u_i 's message. Now, there are at least one u_i left, to which v did not send the *found* message back. If there is exactly one such u_i , v forwards its message; otherwise, v takes the following actions. Let u_s be the node with lowest rank among the rest of the u_i s (i.e., those u_i s which were not sent

a *found* message by v), and u_t , with $t \neq s$, be an arbitrary node among the rest of u_i 's. Now, it must be the case that $\rho_s < \rho_t$ (otherwise, v would send a *found* message to u_s), i.e., u_s is in an earlier phase than u_t . This can happen if in some previous phase, u_t exhausted its ρ -value with smaller λ -value leading to a smaller number of rounds in that phase and a quick transition to the next phase. Node v forwards *explore* message of u_s only and sends back *wait* messages to all u_t ($t \neq s$). A node after receiving a *wait* message simply repeats its exploration phase with the same ρ and λ .

Suppose a node u_i found a higher ranked node u_j via the path $Q(u_i, u_j)$. If u_i 's nearest node of higher rank is u' , then $w(Q) \leq 4d(u_i, u')$. Assume that u_j is found when u_i explored the (ρ, λ) -neighborhood for some ρ and λ . Then $d(u_i, u') > \rho/2$, otherwise, u_i would find u' as a node of higher rank in the previous phase and would not explore the ρ -neighborhood. Now, u_j could be found by u_i in two ways. i) The *explore* message originated by u_i reaches u_j and u_j sends back a *found* message. In this case, $w(Q) \leq \rho$. ii) Some node v receives two *explore* messages originated by u_i and u_j via the paths $R(u_i, v)$ and $S(u_j, v)$ respectively, where $r(u_i) < r(u_j)$ and $w(S) \leq \rho$; and v (on behalf of u_j) sent a *found* message to u_i . In this case, $w(Q) = w(R) + w(S) \leq 2\rho$, since $w(R) \leq \rho$. Thus, in both cases, we have $w(Q) \leq 4d(u_i, u')$.

The cost of the NNT produced by this algorithm is at most $4\lceil \log n \rceil * c(MST)$. The time complexity of the algorithm is $O(D + L \log n)$ and the message complexity is $O(|E| \log L \log n)$. For more details of this algorithm, the proof of correctness, and the analysis of approximation ratio, time complexity, and message complexity, readers are referred to [24].

3.4 Multicast Routing

We can view multicasting as a generalization of broadcasting, where the message has to be sent to only a (required) subset of nodes. This formulation leads to the *minimum cost Steiner tree* problem, a generalization of the MST problem (see also Chapter 7). The Steiner tree connects the required set of nodes and can possibly use other nodes in the graph (these are called Steiner nodes). Finding the minimum cost Steiner tree in a arbitrary weighted graph is NP-hard. The following approximation algorithm is well-known (e.g., [50]). Transform the given graph $G = (V, E, w)$ as a *complete metric* graph $G' = (V, E', w')$ where the pairwise edge costs in E' are the corresponding pairwise (weighted) shortest path distances in G , i.e., $w'(u, v) = dist(u, v)$, where $dist(u, v)$ is the shortest path distance between u and v in G . It can be shown that an MST T' on the required set of nodes in G' is a 2-approximation to the optimal Steiner tree in G . To get a 2-approximate Steiner tree in G , we replace the edges in T' by the corresponding shortest paths in G .

This may possibly create cycles, so some edges may have to be deleted.

Thus distributed algorithms for minimum-cost Steiner tree are at least as involved as those for the MST problem. Distributed approximation algorithms have been recently proposed for this problem. Based on the MST approximation described above, an $O(n \log n)$ -time 2-approximate distributed algorithm has been proposed in [9]. The NNT approach of [24] described in Section 3.3.2 yields a faster $O(\log n)$ -approximation with the same time bounds. Another advantage of the NNT approach is that this algorithm works with virtually no change. Indeed the basic algorithm is: every required node chooses a unique rank and connects to its closest required node of higher rank.

4 Queuing

Queuing theory is the primary methodological framework for analyzing network delay. We study two different approaches here, the first characterized by stochastic modeling of arrival and service times (Section 4.1), and the second by a more general adversarial model (Section 4.3).

4.1 Stochastic Queuing Models

In a typical queueing system, scenario is characterized by a variable set of customer(s) which contend for (limited) resources served by server(s). A customer leaves the system once it receives the service. We note that integral to queueing situations is the idea of uncertainty in, for example, interarrival times and service times. Hence, stochastic models are used to model queueing systems.

Our study of queueing is basically motivated by its use in the study of communication systems and computer networks. Nodes, especially, routers and switches in a network may be modeled as individual queues. The whole system may itself be modeled as a queueing network providing the required service to the data that need to be carried. Queueing theory provides the theoretical framework for the design and study of such networks.

For a single queueing system where customers are identified based on their arrival order, we denote the

following:

$N(t)$ = number of customers in the queue at time t .

$\alpha(t)$ = number of customers who arrived in $[0, t]$.

T_i = time spent in the system by customer i .

Assume that the following three limits exist:

$$N = \lim_{t \rightarrow \infty} \frac{1}{t} \int_0^t N(t) \quad (16)$$

$$\lambda = \lim_{t \rightarrow \infty} \frac{\alpha(t)}{t} \quad (17)$$

$$T = \lim_{t \rightarrow \infty} \frac{\sum_{i=1}^{\alpha(t)} T_i}{\alpha(t)} \quad (18)$$

N is called the steady-state time average of the number of customers in the system, λ is the steady-state arrival rate of customers, and T the steady state time average customer delay. These three quantities are related by the following basic equation known as *Little's Lemma* holds: $N = \lambda T$ (see [46] for a proof).

The $M/M/1$ System

The name $M/M/1$ reflects the standard queueing theory nomenclature whereby: (1) The first letter indicates the nature of the arrival process. For example, M stands for memoryless, i.e., a Poisson arrival process, G stands for a general distribution of interarrival times, D stands for deterministic interarrival times; (2) The second letter indicates the nature of the probability of the server process. For example, M stands for memoryless, i.e., an exponential service time model; G for general, and D for deterministic; and (3) The last number indicates the number of servers (can range from 1 to ∞). The queue discipline that we will assume (unless otherwise stated) is First-in-first-out (FIFO). This specifies how to determine which customer to serve next from among the set of customers waiting for service. We will assume a Poisson arrival process throughout; this makes the models amenable to precise analysis.

The $M/M/1$ system is a simple and basic queueing system where there is a single queue with a single server. Customers arrive according to a Poisson process with rate λ . The probability distribution of the service time is exponential with mean $\frac{1}{\mu}$ sec. The successive interarrival times and service times are statistically independent of each other.

Using a Markov chain analysis, the following properties of the M/M/1 queuing system can be shown (see for e.g., [5]):

1. The expected number of customers in the system at steady state is $N = \frac{\lambda}{\mu - \lambda}$.
2. The average delay per customer is (by Little's Lemma) $T = N/\lambda = \frac{1}{\mu - \lambda}$.
3. The average time spent by the packet waiting in the queue (not including the transmission time) is $W = T - 1/\mu$.
4. The average number of customers in the queue is $N_Q = \lambda W = \frac{\rho^2}{1 - \rho}$, where $\rho = \lambda/\mu$ is called the *utilization factor*.

We look at an application of M/M/1 queues to communication networks. We assume that packets arrive according to a Poisson process to an edge and the packet transmission rate is exponentially distributed. Interarrival and service times are independent, i.e., the length of the arriving packet does not affect the arrival time of the next packet (this is not true in reality when you have a network of queues, see [5]).

Statistical multiplexing verses time-division multiplexing: These are two basic schemes of multiplexing traffic in a communication link. In statistical multiplexing, the packets from different traffic streams are merged into a single queue and transmitted on a first-come first-serve basis. In time division multiplexing (TDM), with m traffic streams, the edge capacity is essentially subdivided into m parts and each part is allocated to one traffic stream.

Given m identical Poisson process each with arrival rate λ/m , which scheme gives better delay results? We assume that the average packet service time is $1/\mu$. Then statistical multiplexing gives a delay corresponding to an arrival rate λ and service rate μ i.e. $T = \frac{1}{\mu - \lambda}$. On the other hand, TDM, i.e., transmitting through m separate channels, gives a much bigger per packet delay $T' = \frac{m}{\mu - \lambda}$ since the service rate per channel now reduces to $\frac{\mu}{m}$. This is the reason why in **packet-switched networks** such as the Internet, where traffic is mostly irregular and bursty, statistical multiplexing is used. In **circuit-switched networks**, such as telephone networks, where each traffic stream is more or less “regular” (as opposed to Poisson), TDM is preferred. The reason being that there will be no waiting time in the queue if such a stream is transmitted on a dedicated line.

M/G/1 System

In this system we have a single server, an infinite waiting room, exponentially distributed interarrival times (with parameter λ) and an arbitrary service time distribution, for which at least the mean value and the standard deviation is known. The service discipline is FIFO. Let X_i is the service time of the i th customer.

X_i 's are identically distributed, mutually independent, and independent of interarrival times.

$E[X] = 1/\mu =$ average service time.

$E[X^2] =$ second moment service time.

The expected waiting time in queue for a customer in this system is given by the Pollaczek-Khintchine formula (P-K formula).

P-K formula: The expected waiting time in queue in a stable M/G/1 system is

$$W = \frac{\lambda E[X^2]}{2(1 - \rho)} \quad (19)$$

where, $\rho = \lambda/\mu$. See [5] for a proof. Using Little's law, the expected number of customers in the queue is

$$N_Q = \lambda W = \frac{\lambda^2 E[X^2]}{2(1 - \rho)} \quad (20)$$

M/G/ ∞ System

In M/G/ ∞ system, arrivals are Poisson with rate λ , however, service times are assumed to be independent with distribution G . i.e., the service of a customer is independent of the services of the other customers and of the arrival process.

Let, $X(t)$ denote the number of customers arrived in the system till time t (some or all of them might have already left the system). Conditioning $\Pr(N(t) = j)$ over $X(t)$ we get:

$$\Pr(N(t) = j) = \sum_{n=0}^{\infty} \Pr(N(t) = j | X(t) = n) e^{-\lambda t} \frac{\lambda^n}{n!}, \quad (21)$$

because, $\Pr(X(t) = n) = e^{-\lambda t} \frac{\lambda^n}{n!}$ due to Poisson arrival with rate λ .

The probability that a customer who arrives at time x will still be present at t is $1 - G(t - x)$.

Let p be the probability that an arbitrary customer that arrives during the interval $[0, t]$ is still in the network at time t . We know, for a Poisson distribution, given that an arrival happens at time interval $[0, t]$, then the arrival is uniform in this interval. Therefore, $\frac{dx}{t}$ is the probability that an arbitrary customer arrives in the interval dx . Thus,

$$p = \int_0^t (1 - G(t - x)) \frac{dx}{t} = \int_0^t (1 - G(x)) \frac{dx}{t}, \quad (22)$$

independently of the others. Then,

$$P(N(t) = j | X(t) = n) = \begin{cases} \binom{n}{j} p^j (1-p)^{n-j} & : j = 0, 1, \dots, n \\ 0 & : j > n \end{cases} \quad (23)$$

Thus,

$$\Pr(N(t) = j) = \sum_{n=j}^{\infty} \binom{n}{j} p^j (1-p)^{n-j} e^{-\lambda t} \frac{\lambda t}{n!} \quad (24)$$

$$= e^{-\lambda t} \frac{(\lambda t p)^j}{j!} \sum_{n=j}^{\infty} \frac{(\lambda t (1-p))^{n-j}}{(n-j)!} \quad (25)$$

$$= e^{-\lambda t p} \frac{(\lambda t p)^j}{j!} \quad (26)$$

Hence we find that, $N(t)$ is Poisson distributed with mean $\lambda \int_0^t (1 - G(x)) dx$.

Application to P2P Networks

A Peer-to-Peer (P2P) network (cf. Section 7) can be modeled as an $M/M/\infty$ system.

The arrival of new nodes is Poisson distributed with rate λ . The duration of time a node stays connected to the network is independently and exponentially distributed with parameter μ . Without loss of generality, let $\lambda = 1$ and let $N = 1/\mu$.

Theorem 4.1 [38] *If $\frac{t}{N} \rightarrow \infty$ then with high probability, $N(t) = N \pm o(N)$.*

Proof: The number of nodes in the network at time t , $N(t)$, is Poisson distributed with mean $\lambda \int_0^t (1 - G(x)) dx$ which is $N(1 - e^{-t/N})$.

If $t/N \rightarrow \infty$, $E[N(t)] = N + o(N)$.

Now, using a Chernoff bound for the Poisson distribution [1], for $t = \Omega(N)$, and for some constants b and $c > 1$:

$$\Pr\left(N(t) - E[N(t)] \leq \sqrt{bN \log N}\right) \geq 1 - 1/N^c. \quad (27)$$

□

4.2 Communication Networks Modeled as Network of Queues

So far we have only looked at a single stand alone queueing system. However, most real systems are better represented as a network of queues. An obvious example is the Internet, where we can model each outgoing link of each router as a single queueing system, and where an end-to-end path traverses a multitude of intermediate routers. In a queueing network a customer finishing service in a service facility is immediately proceeding to another service facility or she is leaving the system.

When we look at a network of queues, the analysis of the system becomes much more complex. The main reasons are:

- Even if the input queue is an $M/M/1$ queue, this is not true for any internal node.
- The inter-arrival times in the downstream queues are highly correlated with the service times of the upstream queues.
- Service times of the same packet in different queues are not independent.

To illustrate with a simple example, consider two edges of equal capacity in tandem connecting two queues, 1 and 2, with the first queue feeding the second. Assume that packet arrive according to a Poisson process with independent exponentially distributed packet lengths. Then the first queue behaves as an $M/M/1$, but the second queue does not. This is because the interarrival times at the second queue are strongly correlated with the packet lengths: longer packets will wait less at the second queue than short packets, since their transmission time at the first queue takes longer, thereby giving more time for the second queue to empty out.

To deal with this situation, Kleinrock [25] introduced the following *independence assumption*:

Adopt an $M/M/1$ model for each communication link regardless the interaction of the traffic on this link with traffic on other links.

Although, the Kleinrock independence assumption is typically violated in real networks, the cost function, that is based on the Kleinrock assumption, represents a useful measure of performance in practice. This turns out to be a reasonable approximation for networks involving Poisson stream arrivals at the entry points, packet lengths that are nearly exponentially distributed, a densely connected network, and moderate-to-heavy traffic loads.

One basic classification of queueing networks is the distinction between open and closed queueing networks. In an open network new customers may arrive from outside the system (coming from a conceptually infinite population) and later on leave the system. In a closed queueing network the number of customers is fixed and no customer enters or leaves the system. An example for an open queueing network may be the Internet, where new packets arrive from outside the system (in fact, from the users).

4.2.1 Jackson's Theorem

Jackson's theorem is an important result in the analysis of a network of queues which shows that under certain conditions a network of queues behaves as a collection of $M/M/1$ queues.

Assume a system of K FIFO queues.

Let $n_i(t)$ be the number of customers at queue i at time t .

Let $n(t) = (n_1(t), \dots, n_K(t))$ be the state of the system at time t .

The steady state probability of the system being in state (n_1, \dots, n_K) is denoted as $P(n)$. From this steady state probability we can derive the marginal probability $P_i(n_i)$ that the node i contains exactly n_i customers. In some cases it is possible to represent the state probabilities as follows:

$$P(n) = \frac{1}{G(n)} P_1(n_1) \times \dots \times P_K(n_K) \quad (28)$$

where $G(n)$ is the so-called normalization constant (it depends on the number of customers in the system). In the case of an open queueing network we have always $G(n) = 1$, in the case of a closed queueing network $G(n)$ must be chosen such that the normalization condition $\sum P(n) = 1$ holds. The above equation represents a *product form* solution. A nice property of this equation is that we can decompose the system and look at every service center separately.

The theorem of Jackson specifies the conditions, under which a product form solution in open queueing networks exist. These conditions are the following:

- The number of customers in the network is not limited.
- New customers arrive at queue i according to a Poisson process with rate r_i (some or all r_i 's may be 0).
- Service at queue i is exponential with rate μ_i .

- A customer can leave the system from any node (or a subset).
- When a customer is served at queue i it proceeds with probability P_{ij} to queue j , and with probability $1 - \sum_j P_{ij}$ it leaves the system.
- All choices are independent of previous choices.
- In every node the service discipline is FIFO.

Let λ_i be the arrival rate to queue i .

$$\lambda_i = r_i + \sum_{j=1}^K \lambda_j P_{ji}, \quad i = 1, \dots, K. \quad (29)$$

Assume that the above system has a unique solution and let $\rho_i = \lambda_i / \mu_i$. Then Jackson's theorem can be stated as:

Theorem 4.2 *If in an open network the condition $\rho_i < 1$ holds for $i = 1, \dots, K$, then under the above mentioned conditions the system has a steady state (n_1, \dots, n_K) and the steady state probability of the network can be expressed as the product of the state probabilities of the individual nodes: $\lim_{t \rightarrow \infty} P_t(n) = P(n) = P_1(n_1) \times \dots \times P_K(n_K)$ and $P_j(n_j) = \rho_j^{n_j} (1 - \rho_j)$ $n_j \geq 0$.*

The theorem implies that the system operates as a collection of independent M/M/1 queues, though the arrival process to each of the queues is not necessarily Poisson.

An application of Jackson's theorem can be seen in routing on the *butterfly network* ([30][Chapter 3.6]) with Poisson arrivals to the inputs, and exponential transition time through the edges.

Assume that the arrival rate of packets to each of the inputs is λ .

Then the arrival rate to each of the internal nodes is λ .

Let $\mu > \lambda$, i.e., $\rho < 1$. Then the system is stable. Since for a n input butterfly network there are $\log n$ links (queues) from any input node to an output node, the expected time in the system for a customer (packet) is $\leq \frac{\log n}{\mu - \lambda}$.

4.3 Adversarial Queuing

Adversarial queuing theory [8] addresses some of the restrictions inherent in probabilistic analysis and queuing theory based on time-invariant stochastic generation. This theory is aimed to systemic study of

queuing with little or no probabilistic assumption. Adversarial queuing is based on an adversarial generation of packets in dynamic packet routing, where packets are injected continuously into the network.

The adversarial model. A **routing network** is a directed graph. Time proceeds in discrete time steps. A packet must travel along a path from its source to its destination. When a packet reaches its destination, it is absorbed. During each step, a packet may be sent from its current node along one of the outgoing edges from that node. At most one packet may travel along any edge of the network in a step. A packet that wishes to travel along any edge at a particular time step but is not sent, can wait in a queue for that edge. The delay of a packet is the number of steps which the packet spends waiting in queues. At each step, an adversary injects a set of packets and specifies the route for each packet. The route of each packet is fixed at the time of injection (nonadaptive routing). The following *load condition* is imposed: Let τ be any sequence of w consecutive time steps (where w is some positive integer) and $N(\tau, e)$ the number of packets injected by the adversary during time interval τ that traverse edge e . Then, for every sequence τ and for every edge e , a (w, ρ) -adversary injects new packets with $N(\tau, e)/w \leq \rho$. ρ is called the rate of injection with window size w . Clearly, if $\rho > 1$, an adversary would congest some edge and the network would be unstable.

If there are more than one packet waiting in the queue of an edge e , a *scheduling policy* determines which packet is to be moved along edge e in each time step. A scheduling policy is greedy if it moves a packet along an edge whenever there is a packet in the queue. A network system is *stable* if there is a finite number M such that the number of packets in any queue is bounded by M . The following theorem shows a general stability result for adversarial queuing.

Theorem 4.3 *In a directed acyclic graph, with any greedy scheduling and any adversarial packet generation with rate $\rho = 1$, a network system is stable.*

Proof: Let $Q_t(e)$ be the number of packets in the queue of edge e at time t and $A_t(e)$ the number of packets that have been injected into the system by time t , are still in the system (not absorbed yet), and are destined to cross edge e . Clearly $Q_t(e) \leq A_t(e)$. Since the injection rate is 1, there exists some window size w such that for any window of time $(t - w, t]$ and for any edge e , the adversary can inject at most w packets. Then for any t' such that $t - w < t' < t$,

$$A_{t'}(e) \leq A_{t-w}(e) + w. \quad (30)$$

For an edge e , let e_1, e_2, \dots, e_k be the edges entering the tail of e . A function ψ is defined as follow.

$$\psi(e) = \max\{2w, Q_0(e)\} + \sum_{i=1}^k \psi(e_i) \quad (31)$$

The theorem is proved by showing that for all $t = lw \geq 0$ ($l \geq 1$) and for all edges e ,

$$A_t(e) \leq \psi(e) \quad (32)$$

Since the graph is directed acyclic, it is easy to see that $\psi(e)$ is finite and is a function of the number of nodes in the network; but it does not depends on time t . Thus, as $Q_t(e) \leq A_t(e)$, Inequality 32 gives stability of the network.

Inequality 32 is proved by induction on l . The claim holds when $l = 0$. Now assume $t = lw$ for $l \geq 1$. Consider the following two cases.

Case 1. $A_{t-w}(e) \leq w + \sum_{i=1}^k \psi(e_i)$:

$$A_t(e) \leq A_{t-w}(e) + w \leq 2w + \sum_{i=1}^k \psi(e_i) \leq \psi(e). \quad (33)$$

Case 2. $A_{t-w}(e) > w + \sum_{i=1}^k \psi(e_i)$:

By induction hypothesis, $A_{t-w}(e_i) \leq \psi(e_i)$. Further, notice that $A_{t-w}(e)$ is at most $Q_{t-w}(e) + \sum_{i=1}^k A_{t-w}(e_i)$.

Thus,

$$Q_{t-w}(e) \geq A_{t-w}(e) - \sum_{i=1}^k A_{t-w}(e_i) \quad (34)$$

$$\geq w + \sum_{i=1}^k \psi(e_i) - \sum_{i=1}^k A_{t-w}(e_i) \quad (35)$$

$$\geq w. \quad (36)$$

That is, there are at least w packets in the queue of edge e at the beginning of time step $t - w$. Thus, by a greedy scheduling, w packets cross e in the next w time steps; but the adversary can inject at most w packets into the system for edge e . Therefore, $A_t(e) \leq A_{t-w}(e)$. By induction hypothesis, $A_{t-w}(e) \leq \psi(e)$. Thus, $A_t(e) \leq \psi(e)$. \square

If the network contains cycles, the system may not be stable for some scheduling policies such as FIFO

(First In First Out) and LIS (Longest In System – where priority is given to the packets that has been in the network for the longest amount of time). However, some scheduling policies such as FTG (Furthest To Go – where priority is given to the packet that has largest number of edges still to be traversed), are stable in a directed network with cycles and adversarial packet generation with rate 1. For proofs of these claims and further details readers are referred to [8].

Adversarial queuing theory also deals with stochastic adversaries. A (w, ρ) *stochastic adversary* generates packets following some probability distribution such that $E[N(\tau, e)]/w \leq \rho$. The adversary can follow different probability distribution at different time steps. If $\rho = 1$, an adversary can inject zero packets with probability 1/2 and two packets with probability 1/2, which is analogous to a random walk on line $[0, \infty)$, and can make the system unstable; after t steps, the expected queue size is approximately \sqrt{t} . hence it is assume that $\rho < 1$. Further constraint is imposed on the p th moment for $p > 2$: there exist constants $p > 2$ and V such that for all sequence τ of w consecutive time steps and all edge e , $E[N^p(\tau, e)] \leq V$. A network system with arbitrary directed acyclic graph, an arbitrary greedy scheduling, and stochastic adversaries with rate $\rho < 1$ is stable [8].

5 Multiple Access Protocols

Multiple access channels provide a simple and efficient means of communication in distributed networks such as the Ethernet and wireless networks. Ethernet is a broadcast local area network (LAN), i.e., when a station (node) wants to transmit a message to one or more stations, all stations connected to the Ethernet LAN will receive it. Since several nodes share the same broadcast channel and only one message can be broadcast at each step, a multiple access protocol (or a contention resolution protocol) is needed to coordinate the transmissions and avoid collisions. Such a protocol decides when to transmit a message and what to do in the event of a collision.

We assume that one message can be sent at a given time. If more than one message is sent at a given time, no message is received (i.e., a collision occurs). A station can check if the “channel” is free. Note that this does not eliminate collisions. Unfortunately, collisions can still occur, since there is a nonelible delay between the time when a station begins to transmit and the other stations detect the transmission. Hence, if two or more stations attempt to transmit within this window of time, a collision will occur. A station can detect if the message it sent was broadcast successfully. A station’s protocol uses only that station’s history

of success and failure transitions (such a protocol is called *acknowledgment-based*); it has no knowledge about other stations, or even the number of other active stations. In case of a collision, the messages (that did not get sent) are queued at their respective stations for retransmission at some point in the future. It is not a good idea to retransmit immediately, since this would result in another collision. Rather, messages are retransmitted after some delay which is chosen probabilistically. Such a strategy is called a *backoff* strategy defined formally below.

Backoff protocol:

Each station follows the following protocol.

1. The backoff counter b is initially set to 0.
2. While the station queue is not empty do
 - (a) with probability $1/f(b)$ try to broadcast.
 - (b) If the broadcast succeeded $b = 0$, else $b = b + 1$.

In principle, f can be an arbitrary function. We have different classes of backoff protocols, depending on f .

Exponential backoff: $f(b) = 2^b$ (The Ethernet uses $2^{\min[b,10]}$).

Linear backoff: $f(b) = b + 1$.

Polynomial backoff: $f(b) = (b + 1)^\alpha$ (for some constant $\alpha > 1$).

Analysis Model

The key theoretical question is: Under what condition, backoff protocols have good performance? For example, is exponential backoff better than polynomial backoff? To answer this rigorously we need a model and a performance measure. We consider the following model [21].

Time is partitioned into steps. At a beginning of a step each station receives a new message with probability λ_i , for $1 \leq i \leq N$, where N is the number of stations in the system. The arrival of new messages is assumed to be independent over time and among stations. (Note that no assumptions are made about the independence of the state of the system from one time step to the next, or between stations.) The overall arrival rate is defined to be $\lambda = \sum_{i=1}^N \lambda_i$. Arriving messages are added to the end of the queues located at each station. Queues can be of infinite size. The system starts with empty queues. Acknowledgment of

a collision or successful transmission is taking place within one step. Message lengths are assumed to be smaller than the duration of a time step.

The main performance measure is *stability*. Let $W_{avg}(t)$ be the average waiting time of messages (packets) that entered the system at time t . Let $L(t)$ the number of messages in queues at the end of step t . We say that a system is *stable* if $E[W_{avg}(t)]$ and $E[L(t)]$ are both bounded with respect to t . (By Little's Lemma $E[L(t)] = \lambda E[W_{avg}(t)]$ if both exist and are finite).

There are important differences between the above model and real-life, e.g., the Ethernet [21]. For example, in the Ethernet, the backoff counter is never allowed to exceed a specified value $b_{max} = 10$. A problem with placing such a bound is that any protocol becomes unstable for any fixed λ and large enough N [21]. Another difference is that in the Ethernet is that messages are discarded if they are not delivered within a specified amount of time. Discarding messages contributes to stability but at the cost of losing a nonzero fraction of the messages. Also assumptions regarding message arrival distribution, message length, and synchronization may not be fully realistic.

We next state two key results of [21]. The first states that the exponential backoff protocol is unstable for any *finite* $N > 2$ when the arrival rate at each station is λ/N , for $\lambda \geq \lambda_0 \approx 0.6$. Previous results showed similar instability results but assumed infinite number of stations (which renders the result somewhat less realistic). The second states that polynomial backoff protocol is stable for any $\alpha > 1$ and any $\lambda < 1$. This was surprising because if the number of station is infinite, it can be shown that polynomial backoff is unstable [22]. The main approach in showing the above results is by analyzing the behavior of the associated Markov chain. For backoff protocols, with finite number of stations, one can associate every possible configuration of backoff counters and queues $(\mathbf{b}, \mathbf{q}) = \{(b_1, \dots, b_N, q_1, \dots, q_N) | b_i \geq 0, q_i \geq 0 \text{ for } 1 \leq i \leq N\}$ with a unique state of the Markov chain. The initial state is identified with $(0, \dots, 0)$. A potential function method is then used to analyze the evolution of the chain. For showing instability results, for example, the potential function is chosen such that the expected change in potential is at least a fixed positive constant (thus the function grows linearly with time) and this is then shown to imply that the protocol is unstable. For example, the following potential function is used to show that the exponential backoff function is unstable.

Define a potential function of the system at time t :

$$\Phi(t) = C \sum_{i=1}^N q_i + \sum_{i=1}^N 2^{b_i} - N \quad (37)$$

where $C = 2N - 1$, $q_i =$ number of packets queued at station i at time t , and $b_i =$ the value of the backoff counter at station i at time t .

The main thrust of the proof is to show that $E[\Phi(t)]$ increases by a constant δ in each step. This will then imply that the system is unstable because of the following lemma.

Lemma 5.1 *If there is a constant $\delta > 0$ (independent of t) such that for all $t > 0$*

$$E[\Delta\Phi(t)] = E[\Phi(t) - \Phi(t-1)] \geq \delta \quad (38)$$

then the system is unstable.

Proof: A message enters the queue of station i at time t with probability $\lambda_i = \lambda/N$.

The expected wait time of this message is at least $q_i + 2^{b_i}$.

For vectors \bar{q} and \bar{b} , let $P(t, \bar{q}, \bar{b})$ be the probability that at time t , and for some station $i = 1, \dots, N$, the queue of station i has q_i items and the counter of i is b_i .

$$E[W_{avg}(t)] \geq \sum_{q,b} \sum_{i=1}^N \lambda_i (q_i + 2^{b_i}) P(t, \bar{q}, \bar{b}). \quad (39)$$

$$E[\Phi(t)] = \sum_{q,b} \sum_{i=1}^N (Cq_i + 2^{b_i} - 1) P(t, \bar{q}, \bar{b}). \quad (40)$$

Thus,

$$E[W_{avg}(t)] \geq \frac{\lambda}{CN} \Phi(t) \geq t\delta \frac{\lambda}{CN} \Phi(0). \quad (41)$$

$$E[W_{avg}(t)] \rightarrow \infty \text{ as } t \rightarrow \infty. \quad (42)$$

□

Once a protocol is determined to be stable (for a given arrival rate), the next step is to compute the expected delay that a message incurs, i.e., $E[W_{avg}]$, which is an important performance measure, especially for high-speed communication networks. The work of Hastad et al. [21] also showed that long delays are unavoidable in backoff protocols: they showed that the delay of any stable exponential or polynomial backoff protocol is at least polynomial in N .

We next consider a protocol due to Raghavan and Upfal [42] that gives a protocol that guarantees a delay of $O(\log N)$, provided the total arrival rate λ is less than a fixed constant $\lambda' \approx 1/10$. The main feature of this protocol is that each sender has a **transmission buffer** of size $O(\log N)$ and a *queue*. (The protocol assume that each station knows N .) Messages awaiting transmission are stored in the buffer or in the queue. Throughout execution of the protocol a sender is in one of the two states: normal state or a reset state. We need the following definitions. Let μ, α, γ be suitably chosen fixed constants (these are fixed in the proof) [42]. *Count_attempts(s)* keeps a count of the number of times station s tries to transmit a message from its buffer in the $4\mu N \log N$ most recent steps. *Failure_counts(s)* stores the failure rates in transmission attempts of packets from the buffer of s in the most recent $\mu \log N$ attempts. *Random_number()* is a function that returns a random number uniformly chosen in the range $[0, 1]$, independent of the outcomes of previous calls to the function.

Communication protocol for station s :

While in the normal state repeat:

1. Place new messages in the buffer.
2. Let X denote the number of packets in the buffer.
if $Random_number() \leq X/8\alpha \log N$ then
 - (a) Try to transmit a random message from the buffer.
 - (b) Update *Count_attempts(s)* and *Failure_counts(s)*.
else
if $Random_number() \geq 1 - 1/N^2$ the transmit the message at the head of the queue.
3. If (*Count_attempts(s)* $\geq \mu \log n$ and *Failure_counts(s)* $> 5/8$) or if ($X > 2\alpha \log N$) then
 - (a) Move all messages in the buffer to the end of the queue.
 - (b) Switch to the reset state for $4\mu N^2 \log N + \gamma \log N$ steps.

While in the reset state repeat:

1. Append any new messages to the queue.
2. If $Random_number() \leq 1/N^2$ then transmit the message at the head of the queue.

The main intuition behind the protocol is that most of the time, things behave normally, i.e., there is not too many collisions and the buffer size is $O(\log N)$ and the delay is $O(\log N)$. However, there can be bad events (e.g., every station generates a packet in every one of N^{10} consecutive steps), but these happen with very low probabilities. The reset state and the queue are then used as an emergency mechanism to handle such cases. Note that the delay for sending messages in the queue is polynomial in N , but this happens with very low probability, so that the overall expectation is still $O(\log N)$.

Raghavan and Upfal also show that for a class of protocols that include backoff protocols, if the total arrival rate λ is less than some fixed constant $\lambda_1 < 1$, then the delay must be $\Omega(N)$. Using a more sophisticated protocol, Goldberg et al. [20] are able to show a constant expected delay for arrival rate up to $1/e$. This protocol is practical if one assumes that the clocks of the stations are synchronized. If not, it is shown that $O(1)$ delay can still be achieved, but the constant is too large to be practical.

6 Resource Allocation and Flow Control

Congestion and flow control is a key ingredient of modern communication networks (typically implemented in the transport layer) which allows many users to share the network without causing congestion failure. Of course, congestion control can be trivially enforced if users do not send any packets! Thus, in addition to congestion control, we would like to maximize network throughput. It is possible to devise schemes that can maximize network throughput, for example, while denying access to some users. We would like to devise congestion control schemes that operate with high network utilization while at the same time ensuring that all users get a fair share of resources. This can be cast as an optimization problem (defined formally below) and can be solved in a centralized fashion. However, to be useful in a network, we need to devise schemes for solving this *distributively*. The “additive increase/multiplicative decrease scheme” of Internet’s flow control (TCP) (see e.g., [27]) is an attempt to solve the above resource allocation problem. The framework we study next provides a provably good distributed algorithm for optimal resource allocation and flow control and can be viewed as a rigorous generalization of the simple flow control scheme of TCP.

Optimization Problem. Consider a network that consists of a set $L = \{1, \dots, L\}$ links of capacity c_l , $l \in L$. The network is shared by a set $S = \{1, \dots, S\}$ of sources (users). The goal is to share the network resources (i.e., link bandwidths) in a fair manner. Fairness is captured as follows. Each user s has an associated utility function $U_s(x_s)$ (an increasing and strictly concave function) — the utility of the source as

a function of its transmission rate x_s . We assume that a *fair* resource allocation tries to maximize the *sum* of the utilities of all the users in the network. We assume that the route of each source is fixed, i.e, $L(s) \subseteq L$ is a set of links that source s uses. For each link l , let $S(l) = \{s \in S | l \in L(s)\}$ be the set of sources that use link l . Note that $l \in L(s)$ if and only if $s \in S(l)$.

The resource allocation problem can be formulated as the following nonlinear program (e.g., [23, 32]). The objective is to choose source rates $x = \{x_s, s \in S\}$ so as to

$$\max_{x_s \geq 0} \sum_s U_s(x_s)$$

subject to

$$\sum_{s \in S(l)} x_s \leq c_l, l = 1, \dots, L. \quad (43)$$

The constraint says that the aggregate source rate at any link l does not exceed the capacity. If the utility functions are strictly concave, then the above nonlinear program has a unique optimal solution. To solve this optimization problem directly, one should have knowledge of the utility functions and routes of all sources in the network. This requires coordination among all sources which is not practical in large networks such as the Internet. Our goal is a distributed solution, where each source adapts its transmission rate based only on local information. We next describe a distributed solution due to Low and Lapsley [32] that works on the *dual* of the above nonlinear problem.

Dual Problem. To set up the dual, we define the Lagrangian

$$L(x, p) = \sum_s U_s(x_s) - \sum_l p_l \left(\sum_{s \in S(l)} x_s - c_l \right) \quad (44)$$

$$= \sum_s \left(U_s(x_s) - x_s \sum_{l \in L(s)} p_l \right) + \sum_l p_l c_l. \quad (45)$$

$p = \{p_1, \dots, p_L\}$ are the Lagrange multipliers. The first term is separable in x_s , and hence

$$\max_{x_s} \sum_s \left(U_s(x_s) - x_s \sum_{l \in L(s)} p_l \right) = \sum_s \max_{x_s} \left(U_s(x_s) - x_s \sum_{l \in L(s)} p_l \right). \quad (46)$$

The objective function of the dual problem is thus (e.g., see [6]):

$$\min_{p_l \geq 0} D(p) := \sum_s \max_{x_s \geq 0} \left(U_s(x_s) - x_s \sum_{l \in L(s)} p_l \right) + \sum_l p_l c_l. \quad (47)$$

The first term of the dual-objective function $D(p)$ is decomposable into S separable subproblems, one for each source. Furthermore, a key observation that emerges from the dual formulation is that each source can *individually* compute its optimal source rate, without the need to coordinate with other sources, provided it knows the optimal Lagrange multipliers. One can naturally interpret the Lagrange multiplier, p_l , as the price per unit bandwidth at link l . Then $\sum_{l \in L(s)} p_l$ is the total price per unit bandwidth for all links in the path of s . Thus, the optimal rate of a source depends only the aggregate price on its route and can be computed by solving a simple maximization problem:

$$x_s(p) = U_s'^{-1} \left(\sum_{l \in L(s)} p_l \right) \quad (48)$$

where $U_s'^{-1}(\cdot)$ denotes the inverse of the derivative of U_s . Of course, to compute x_s we first need to compute the minimal prices which we focus on next.

The dual problem of computing the optimal prices is solved by using the gradient projection method (see e.g., [6]). This yields the following adjustment rule that progressively converges to the optimal:

$$p_l(t+1) = \max \left(p_l(t) + \left(\gamma \sum_{s \in S(l)} x_s(p(t)) - \gamma c_l \right), 0 \right) \quad (49)$$

where $\gamma > 0$ is the stepsize parameter (has to be sufficiently small for convergence [33]). The above formula has nice interpretation in the language of economics: if the demand $\sum_{s \in S(l)} x_s(p(t))$ for bandwidth at link l exceeds the supply c_l , then raise the price $p_l(t)$; otherwise, reduce price $p_l(t)$. Another key observation is that, given the aggregate source rate $\sum_{s \in S(l)} x_s(p(t))$ that goes through link l , the adjustment rule is completely distributed and can be implemented by individual links (or the routers) using only local information. Equations 48 and 49 suggests the following iterative (and synchronous) distributed algorithm:

(1) Link l 's algorithm: In step $t \geq 1$, a link l receives rates $x_s(t)$ from all sources $s \in S(l)$ ($x_s(0)$ can be zero) that go through link l . It then computes a new price $p_l(t+1)$ ($p_l(0) = 0$) based on Equation 49 and communicates the new price to all sources $s \in S(l)$ that use link l .

(2) Source s 's algorithm: In step $t \geq 1$, a source s receives from the network the aggregate price $\sum_{l \in L(s)} p_l$ of links on its path. It then chooses a new transmission rate $x_s(p(t+1))$ computed by Equation 48 and communicates this new rate to links $l \in L(s)$ in its path.

Low and Lapsley [32] show the convergence of the algorithm when the stepsize parameter is appropriately chosen. They further show the algorithm can be extended to work in an asynchronous setting where the sources and links need not compute in lockstep.

7 Peer-to-Peer Networks

P2P network is a highly *dynamic* network: nodes (peers) and edges (currently established connections) appear and disappear over time. In contrast to a static network, a P2P network has no fixed infrastructure with respect to the participating peers, although the underlying network (Internet) can be static. Thus a P2P network can be considered as an **overlay network** over an underlying network, where the communication between adjacent peers in the P2P network may in fact go through one or more intermediate peers in the underlying network. Peers communicate using only local information since a peer does not have global knowledge of the current topology, or even the identities (e.g., IP addresses) of other peers in the current network.

In a dynamic network, even maintaining a basic property such as connectivity becomes non-trivial. Consider search — a common and pervasive P2P application. Clearly, connectivity is important for reachability of search queries. Having a network of low diameter is also important. Consider P2P systems running the Gnutella protocol [51]. They use a flooding-like routing algorithm: queries fan-out from the source node and the search radius is limited by a “Time to Live (TTL)” parameter (this is set to an initial value at the source node, and is decremented in each hop). Thus a low diameter is not only helpful in enlarging the scope of search but also in reducing the associated network traffic. At the same time, it is desirable to have nodes with bounded degree; this will also help the traffic explosion problem in Gnutella-like networks by reducing the fan-out in each stage.

7.1 A Distributed Protocol for Building a P2P Network

A fundamental challenge is to design distributed protocols that maintain network connectivity and low diameter and which operates without having global knowledge. We next discuss the protocol due to Pandurangan,

Raghavan, and Upfal [38] which is a distributed protocol for constructing and maintaining connected, low-diameter, constant-degree P2P networks. The protocol specifies what nodes a newly arriving node should connect, and when and how to replace lost connections (edges) when an existing node leaves. The model assumes that an incoming node knows the identities (IP addresses) of a small number of existing nodes (at least one such address is needed to get access to the network).

We assume that there is a central element called the *host server* which, at all times, maintains a *cache* containing a list of nodes (i.e., their IP addresses). In particular, assume that the host server maintains K nodes at all times, where K is a constant. The host server is reachable by all nodes at all times; however, it need not know of the topology of the network at any time, or even the identities of all nodes currently on the network. We only expect that (1) when the host server is contacted on its IP address it responds, and (2) any node on the P2P network can send messages to its neighbors.

Before we state the protocol we need some terminology. When a node is in the cache, we refer to it as a *cache node*. It accepts connections from all other nodes. A node is *new* when it joins the network, otherwise it is *old*. The protocol ensures that the degree (number of neighbors) of all nodes will be in the interval $[D, C + 1]$, for two constants D and C .

A new node first contacts the host server, which gives it D random nodes from the current cache to connect to. The new node connects to these, and becomes a *d-node*; it remains a d-node until it subsequently either enters the cache or leaves the network. The degree of a d-node is always D . At some point the protocol may put a d-node into the cache. It stays in the cache until it acquires a total of C connections, at which point it leaves the cache, as a *c-node*. A c-node might lose connections after it leaves the cache, but its degree is always at least D . A c-node has always one *preferred* connection, made precise below. The protocol is summarized below as a set of rules applicable to various situations that a node may find itself in.

P2P protocol for node v :

1. On joining the network: Connect to D cache nodes, which are chosen uniformly at random from the current cache. Note that $D < K$.
2. Reconnect rule: If a neighbor of v leaves the network, and that connection was not a preferred connection, connect to a random node in the cache with probability $D/d(v)$, where $d(v)$ is the degree of v before losing the neighbor.
3. Cache Replacement rule: When a cache node v reaches degree C while in the cache (or if v drops out

of the network), it is replaced in the cache by a d -node from the network. Let $r_0(v) = v$, and let $r_k(v)$ be the node replaced by $r_{k-1}(v)$ in the cache. The replacement d -node is found by the following rule:

$k = 0$;

while (a d -node is not found) **do**

search neighbors of $r_k(v)$ for a d -node;

$k = k + 1$;

endwhile

4. Preferred Node rule: When v leaves the cache as a c -node it maintains a *preferred connection* to the d -node that replaced it in the cache. If v is not already connected to that node this adds another connection to v . Thus the maximum degree of a node is $C + 1$. Also, a c -node can follow a chain of preferred connections to reach a cache node.
5. Preferred Reconnect rule: If v is a c -node and its preferred connection is lost, then v reconnects to a random node in the cache and this becomes its new preferred connection.

Finally, note that the degree of a d -node is always D . Moreover, every d -node connects to a c -node. A c -node may lose connections after it leaves the cache, but its degree is always at least D . A c -node has always one *preferred* connection to another c -node.

7.2 Protocol Analysis

To analyze the protocol, we assume the $M/M/\infty$ model (see Section 4). In other words, we have a “stochastic adversary” which deletes and inserts nodes. (A worst-case adversary is trivial to analyze for this setting, since such an adversary can make sure that the network is always disconnected.) This also turns out to be a reasonable model (it approximates real-life P2P networks quite well [44]). Under this model the steady state size of the network depends only on the ratio $\lambda/\mu = N$ (see Section 4.1). Let G_t be the network at time t (G_0 has no vertices).

7.3 Connectivity

The following theorem shows that the protocol keeps the network connected with large probability at *any* instant of time (after an initial time period of $\Omega(\log N)$).

Theorem 7.1 *There is a constant a such that at any given time $t > a \log N$,*

$$\Pr(G_t \text{ is connected}) \geq 1 - O\left(\frac{\log^2 N}{N}\right).$$

The main idea in the proof of the above theorem is using the preferred connections as a “backbone”: we can show that at all times, each node is connected to some cache node directly or through a path in the network. The “backbone” is kept connected with large probability by incoming nodes: here the “randomness” in choosing the cache node by an incoming node proves crucial. Also, an important consequence from the proof is that the above theorem does not depend on the state of the network at time $t - c \log N$; therefore it can be shown to recover rapidly from fragmentation — a nice “self-correcting” property of the protocol. More precisely we have the following corollary:

Theorem 7.2 *There is a constant c such that if the network is disconnected at time t ,*

$$\Pr(G_{t+c \log N} \text{ is connected}) \geq 1 - O\left(\frac{\log^2 N}{N}\right).$$

Also, if the network is not connected then it has a connected component of size $N(1 - o(1))$.

The theorem also shows that if the network becomes disconnected, then a very large fraction of the network remains connected with high probability.

7.4 Diameter

The key result of [38] is that of the diameter of G_t : It is only logarithmic in the size of the network at any time t with large probability.

Theorem 7.3 *At any any time t (after the network has converged, i.e., $t/N \rightarrow \infty$), with high probability, the largest connected component of G_t has diameter $O(\log N)$. In particular, if the network is connected (which has probability $1 - O(\frac{\log^2 N}{N})$) then with high probability its diameter is $O(\log N)$.*

The high-level idea of the proof is to show that the network at any time resembles a *random graph*, i.e., the classical *Erdos-Renyi* or $G(n, p)$ random graph (n is the number of vertices and p is the probability of having an edge between any two vertices) [7]. The intuition underlying the analysis is that connections are chosen randomly, and thus the topology should resemble a $G(n, p)$ random graph. In a $G(n, p)$ random graph the *connectivity threshold* is $\log n/n$. That is, if p (probability that an edge occurs between two nodes) is greater than $(1 + \epsilon) \log n/n$ then almost surely the graph is connected with $O(\log n)$ diameter; if it is less than $(1 - \epsilon)$ then almost surely the graph will not be connected (for every constant $\epsilon > 0$) [7, 13].

First, it can be shown that there is a very small ($\approx 1/N$) probability that *any* two c -nodes are connected to each other (note that d -nodes are connected to c -nodes); however, this alone is not sufficient to guarantee low diameter since this is below the threshold needed for connectivity. We appeal to a key idea from the theory of $G(n, p)$ graphs: if $p = c/n$, for a suitably large constant c , then almost surely, there is a *giant connected component* in the graph whose size is polynomial in n and has $O(\log n)$ diameter [13, 7]. The protocol of Pandurangan et al. [38] is designed in such a way that a fraction of nodes have a so-called “preferred” connection to a set of random nodes and these connections are always maintained. Using these preferred connections one can (indirectly) show that the giant component that emerges from the random process is indeed the whole graph. A number of works have subsequently used random graphs to efficiently design P2P with good properties, see e.g., [29, 14, 19].

In a dynamic network, there is the additional challenge of quantifying the work done by the algorithm to *maintain* the desired properties. An important advantage of the above protocol is that it takes $O(\log N)$ (N is the steady state network size) work per insertion/deletion. In a subsequent paper, using the same stochastic model, Liben-Nowell, Balakrishnan, and Karger [31] show that $\Omega(\log N)$ work is also required to maintain connectivity. A drawback of the protocol of [38] is that it is not fully decentralized — the entering node needs to connect to a small set of random nodes which is assumed to be available in a centralized fashion. A number of subsequent papers have addressed this and devised fully decentralized protocols which guarantee similar properties [31, 36, 29, 19, 14].

Before we end our discussion on P2P, we briefly mention another important design approach, referred to as a *Distributed Hash Table (DHT)* (see e.g., [43, 47, 36]). A DHT does two things: (1) creates a fully decentralized index that maps file identifiers to peers and (2) allows a peer to search for a file very efficiently (typically logarithmically in the size of the network) without flooding. Such systems have been called as *structured* P2P networks, unlike Gnutella, for example, which are *unstructured*. In unstructured networks, there is no relation between the file identifier and the peer where it resides. Many of the commercially deployed P2P networks are unstructured.

8 Research Issues and Summary

This chapter has focussed on theoretical and algorithmic underpinnings of today’s communication networks, especially the Internet. The topics covered are routing algorithms (including algorithms for broadcast

and multicast), queuing theory (both stochastic and adversarial), multiple access problem, resource allocation/flow control, and peer-to-peer network protocols. We have given only a brief overview of many of these which illustrates the basic theoretical problems involved and the approaches taken to solve them. Tools and techniques from a variety of areas are needed — distributed algorithms, probabilistic models and analysis, graph theoretic algorithms, non-linear optimization to name a few. Section 10 gives pointers to gain background on these tools and techniques, and further reading on these topics.

There are many research challenges. Designing efficient distributed approximation algorithms for fundamental network optimization problems such as minimum Steiner tree and minimum delay routing is an important goal. Designing provably efficient P2P networks remains a key problem at the application layer. The goal is to design provably fast protocols for building and maintaining P2P networks with good topological properties, while at the same time guaranteeing efficient and robust services such as search and file sharing. For these we need efficient distributed algorithm that work well on dynamic networks. Another problem is understanding confluent flows and its impact on routing in the Internet. Developing realistic network models will be a key step in analysis.

9 Defining Terms

Arborescence: An arborescence in a directed graph is a subgraph such that there is unique path from a given root to each other node in the subgraph.

Circuit-switched networks: A communication network where a physical path (channel) is obtained for and dedicated to a single connection between two end-points in the network for the duration of the connection.

Confluent flow: A flow is said to be confluent if all the flows arriving at a node destined for a particular destination departs from the node along a single edge.

Congestion: The congestion of an edge is defined as the total amount of flow going through the edge.

Count to infinity problem: A problem that shows that distance vector routing algorithm can be slow in converging to the correct values in case of a change in link costs.

Diameter: Diameter of a network is the number of links in a (unweighted) shortest path between the furthest pair of nodes.

Distance vector: In some routing algorithms, each node maintains a vector, where the i th component

of the vector is an estimate of its distance to the i th node in the network.

Dynamic network: A network in which the topology changes from time to time.

IP Address: An IP address (Internet Protocol address) is a unique address that the network devices such as routers, host computers, and printers use in order to identify and communicate with each other on a network utilizing the Internet Protocol.

Minimum spanning tree: A minimum spanning tree is a tree spanning all nodes in the networks that has the minimum cost among all possible such trees. The cost of a tree is the sum of the costs of all edges in the tree.

Multiple access problem: The problem of coordinating the simultaneous transmissions of multiple nodes in a common link.

Overlay network: A structured virtual topology over a physical network. A link in the overlay network is a virtual or logical link which corresponds to a path, perhaps through many physical links, in the underlying network.

Packet-switched networks: A communication network where messages are divided into packets and each packet is then transmitted individually. The packets can follow different routes to its destination. Once all the packets forming a message arrive at the destination, they are combined into the original message.

Route: A route of a packet is a sequence of links which the packet travels through in the network.

Routing Network: A routing network is a directed graph where the data packets flow along the edges and in the direction of the edges.

Scheduling policy: A scheduling policy is a method of selecting a packet to be forwarded next along an edge from the packets waiting in the queue of that edge.

Shortest path or Least-cost path: The path between source and destination that has the minimum cost. The cost of a path is the sum of the costs of the links in the path.

Shortest path routing: Finding a shortest path between the source and destination and routing the packets through the links along the shortest path.

Splittable flow: A flow that need not be confluent.

Transmission buffer: A buffer in a node to temporarily store the packets that are ready for transmission.

References

- [1] Alon, N. and Spencer, J., *The Probabilistic Method*, 2nd edition, John Wiley, 2000.
- [2] Awerbuch, B., Bar-Noy, A., and Gopal, M., Approximate Distributed Bellman-Ford Algorithms, *IEEE Transactions on Communications*, **42**(8), 1994, 2515-2517.
- [3] Awerbuch, B. and Leighton, F., A Simple Local-control Approximation Algorithm for Multicommodity Flow, *Proceedings of the 34th IEEE Symp. on Foundations of Computer Science (FOCS)*, 1993, 459-468.
- [4] Awerbuch, B. and Peleg, D., Network Synchronization with Polylogarithmic Overhead, *31st IEEE FOCS*, 1990, 514-522.
- [5] Bertsekas, D. and Gallager, R., *Data Networks*, 2nd edition, Prentice Hall, 1992.
- [6] Bertsekas, D. and Tsitsiklis, J., *Parallel and Distributed Computation*, Prentice Hall, Englewood Cliffs, NJ, 1989.
- [7] Bollobas, B., *Random Graphs*, Cambridge University Press, 2001.
- [8] Borodin, A., Kleinberg, J., Raghavan, P., Sudan, M., and Williamson, D., Adversarial Queuing Theory, *Journal of the ACM*, Vol. 48, No. 1, Pages 13–38, Jan. 2001.
- [9] Chalermsook, P. and Fakcharoenphol, J., Simple Distributed Algorithms for Approximating Minimum Steiner Trees, *Proceedings of International Computing and Combinatorics Conference (COCOON)*, 2005, 380-389.
- [10] Chen, J., Kleinberg, R.D., Lovasz, L., Rajaraman, R., Sundaram, R., and Vetta, A., (Almost) tight bounds and existence theorems for confluent flows, *Proceedings of STOC*, 2004, 529-538.
- [11] Chen, J., Marathe, M., Rajmohan, R., and Sundaram, R., The Confluent Capacity of the Internet: Congestion vs. Dilation, *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2006.
- [12] Chen, J., Rajmohan, R., and Sundaram, R., Meet and Merge: Approximation Algorithms for Confluent Flows, *Journal of Computer and System Sciences*, **72**, 2006, 468-489.

- [13] Chung, F. and Lu, L., Diameter of Random Sparse Graphs, *Advances in Applied Math.*, **26**, 257-279, 2001.
- [14] Cooper, C., Dyer, M., and Greenhill, C., Sampling regular graphs and a peer-to-peer network, in *Proc. of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, New York–Philadelphia (2005), pp. 980-988.
- [15] Cormen, T., Leiserson, C., Rivest, R., and Stein, C., *Introduction to Algorithms*, MIT Press and McGraw Hill, 2001.
- [16] Elkin, M., Unconditional lower bounds on the time-approximation tradeoffs for the distributed minimum spanning tree problem, *Proc. of the ACM Symposium on Theory of Computing*, Chicago, IL, 2004, pages 331 – 340.
- [17] Gallager, R., A Minimum Delay Routing Algorithm Using Distributed Computation, *IEEE Transactions on Communications*, **25**(1), 1997, 73-85.
- [18] Gallager, R., Humblet, P., and Spira, P., A Distributed Algorithm for Minimum-Weight Spanning Trees, *ACM Transactions on Programming Languages and Systems*, **5**(1), 1983, pages 66–77.
- [19] Gkantsidis, C., Mihail, M., and Saberi, A., On the Random Walk method in Peer-to-Peer Networks, *Proc. INFOCOM*, 2004.
- [20] Goldberg, L., Mackenzie, P., Paterson, M., and Srinivasan, A., Contention Resolution with Constant Expected Delay, *JACM*, **47**(6), 1048-1096, 2000.
- [21] Hastad, J., Leighton, T., and Rogoff, B., Analysis of Backoff Protocols for Multiple Access Channels, *SIAM J. Comput.*, **25**(4), 1996, 740-774.
- [22] Kelley, F., Stochastic models of Computer Communication Systems, *J. Roy. Statist. Soc. Ser. B*, **47**, 1985, 379-395.
- [23] Kelly, F., Maulloo, A., Tan, D., Rate Control for Communication Networks: Shadow Prices, Proportional Fairness and Stability, *J. Oper. Res. Soc.*, **49**(3), 1998, 237-252.

- [24] Khan, M. and Pandurangan, G., A Fast Distributed Approximation Algorithm for Minimum Spanning Trees, *Distributed Computing*, **20**, 2008, 391-402. Conference version: *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, 2006, LNCS 4167, Springer-Verlag, 2006, 355-369.
- [25] Kleinrock, L., *Communication Nets: Stochastic Message Flow and Delay*, McGraw-Hill, 1964.
- [26] Kleinrock, L., *Queueing Systems, Vol. 1 and 2*, Wiley, 1975 and 1976.
- [27] Kurose, J.F. and Ross, K.W., *Computer Networking: A Top-Down Approach*, 4th Edition, Addison Wesley, 2008.
- [28] Kutten, S. and Peleg D., Fast distributed construction of k-dominating sets and applications, *Journal of Algorithms*, Vol. 28, 1998, pages 40–66.
- [29] Law, C. and Siu, K., Distributed Construction of Random Expander Networks, *IEEE INFOCOM*, 2003.
- [30] Leighton, F., *Introduction to Parallel Algorithms and Architectures*, Morgan Kaufmann, 1992.
- [31] Liben-Nowell, D., Balakrishnan, H., and Karger, D., Analysis of the Evolution of Peer-to-Peer Systems. *ACM Conf. on Principles of Distributed Computing (PODC)*, Monterey, CA, July 2002.
- [32] Low, S. and Lapsley, D., Optimization Flow Control-I: Basic Algorithm and Convergence, *IEEE/ACM Transactions on Networking*, **7**(6), 1999, 861-874.
- [33] Low, S. and Srikant, R., A Mathematical Framework for Designing a Low-Loss, Low-Delay Internet, *Networks and Spatial Economics*, Kluwer Academic Publishers, **4**(1), 2004, 75-101.
- [34] Lua, E., Crowcroft, J., Pias, M., Sharma, R., and Lim, S., A Survey and Comparison of Peer-to-Peer Overlay Network Schemes, *IEEE Communications Surveys & Tutorials*, **7**(2), 2005, 72-93.
- [35] Lynch, N., *Distributed Algorithms*, Morgan Kaufmann Publishers, 1996.
- [36] Malkhi, D., Naor, M., and Ratajczak, D., Viceroy: A Scalable and Dynamic Emulation of the Butterfly, *ACM Principles of Distributed Computing*, 2002.
- [37] Motwani, R. and Raghavan, P., *Randomized Algorithms*, Cambridge University Press, 1995.

- [38] Pandurangan, G., Raghavan, P., and Upfal, E., Building Low Diameter Peer-to-Peer Networks, *IEEE Journal on Selected Areas in Communications*, **21**(6), Aug. 2003. Conference version in *Proceedings of the 42nd Annual IEEE Symposium on the Foundations of Computer Science (FOCS)*, 2001, 492-499.
- [39] Park, K., The Internet as a Complex System, in *The Internet as a Large-Scale Complex System*, K. Park and W. Willinger (Eds.), Sante Fe Institute, Studies in the Sciences of Complexity, Oxford University Press, 2005.
- [40] Peleg, D., *Distributed Computing: A Locality Sensitive Approach*, SIAM, 2000.
- [41] Peleg, D. and Rabinovich, V., A near-tight lower bound on the time complexity of distributed MST construction, *Proc. of the 40th IEEE Symp. on Foundations of Computer Science*, 1999, pages 253–261.
- [42] Raghavan, P. and Upfal, E., Stochastic Contention Resolution with Short Delays, *SIAM J. Comput.*, **28**(2), 1998, 709-719.
- [43] Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S., A Scalable Content-Addressable Network in *Proceedings of ACM SIGCOMM*, 2001.
- [44] Saroiu, S., Gummadi, P.K., and Gribble, S.D., A Measurement Study of Peer-to-Peer File Sharing Systems, in *Proceedings of Multimedia Computing and Networking (MMCN)*, San Jose, 2002.
- [45] Scheideler, C., *Universal Routing Strategies for Interconnection Networks*, LNCS 1390, Springer-Verlag, 1998.
- [46] Stidham, S., A Last Word on $L = \lambda W$, *Oper. Res.*, **22**, 417-421.
- [47] Stoica, I., Morris, R., Karger, D., Kaashoek, M., and Balakrishnan, H., Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications, in *Proceedings of ACM SIGCOMM*, 2001.
- [48] Tarjan, R., *Data Structures and Network Algorithms*, SIAM, 1987.
- [49] Tel, G., *Introduction to Distributed Algorithms*, Second Edition, Cambridge University Press, 2000.
- [50] Vazirani, V., *Approximation Algorithms*, Springer-Verlag, 2001.
- [51] The Gnutella Protocol Specification v0.4. http://www9.limewire.com/developer/gnutella_protocol.0.4.pdf

10 Further Information

A good introduction to computer networks and the Internet is [27]. Bertsekas and Gallager's book [5] is an introduction to communication networks blending theory and practice. Kleinrock's books [26] are standard references for Queuing theory. Further information on adversarial queuing theory can be found in [8]. References for distributed algorithms include [35, 49]. The book by Peleg [40] focuses on locality aspects of distributed computing. A survey of various P2P schemes can be found in [34]. For more details on resource allocation and congestion control, we refer to the survey of Low and Srikant [33].

Some topics that we have not covered here include topology models for the Internet, traffic modeling, and Quality of Service. For a survey of these issues we refer to [39]. There is large body of work concerning packet routing in tightly coupled interconnection networks (parallel architectures such as trees, meshes, and hypercubes). Leighton's book [30] is a good reference for this. Scheideler's book [45] deals with universal routing algorithms i.e., algorithms that can be applied to arbitrary topologies.

Theory of communication networks is one of the most active areas of research with explosive growth in publications in the last couple of decades. Research gets published in many different conferences. Outlets for theoretical aspects of distributed computing and networks include *ACM Symposium on Principles of Distributed Computing (PODC)*, *International Symposium on Distributed Computing (DISC)*, *ACM Symposium on Parallel Algorithms and Architectures*, and *International Conference on Distributed Computing Systems (ICDCS)*. Traditional theory conferences such as *ACM Symposium on Theory of Computing (STOC)*, and *IEEE Symposium on Foundation of Computer Science (FOCS)*, and *ACM-SIAM Symposium on Discrete Algorithms (SODA)* also contain papers devoted to theoretical aspects of communication networks. The *IEEE Conference on Computer Communications (INFOCOM)* conference covers all aspects of networking, including theory.