

## Chapter 5 *Topology design and analysis*

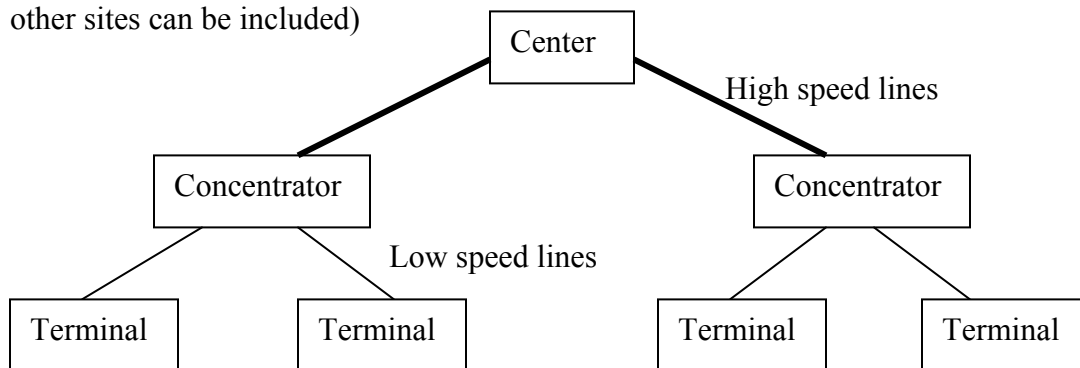
### Topics covered:

Topology design. Network design algorithms. Terminal assignment. Concentrator location. Traffic flow analysis and performance evaluation. Network reliability.

## 5.1 Topology design

### 5.1.1 Centralized Network design

- **Centralized network:** is where all communication is to and from a single central site.
- The “central site” is capable of making routing decisions.  
→ Tree topology provides only one path through the center (For reliability, lines between other sites can be included)



- Three different problems:
  - **Multipoint line topology:** selection of links connecting terminals to concentrators or directly to the center.
  - **Terminal assignment:** association of terminals with specific concentrators.
  - **Concentrator location:** deciding where to place concentrators, and whether or not to use them at all.

### 5.1.2 Finding Trees in Graphs

- Used to design and analyze networks.
- Connect a number of nodes to a central node:
  - **Node:** Hub, Switch, Router, etc.
  - **Central node:** backbone

- A tree is a graph with no loops, with only one path between any pair of nodes.
- Trees are minimal networks: provide connectivity without any unnecessary additional links:
  - Minimally reliable and robust
  - Networks are more highly connected (but design starts with a tree)

### 5.1.2.1 Tree Traversals

- Visit all nodes in a tree: edges are traversed twice.
- First, identify a node as the root
- Assume the tree is directed (outward from the root)
- Two algorithms:
  - BFS (Breadth First Search):
    - Nodes closest to root are visited first
    - Implemented using a queue (FIFO)
  - DFS (Depth First Search):
    - Visits an unvisited neighbor of the node just visited.
    - Implemented using a stack (LIFO)
- Both traversals (BFS and DFS) can be preorder traversals (i.e., visit nodes then successors) or post-order traversals (i.e., successors visited first).
- Traversal is generalized to undirected graphs by keeping track of which nodes were visited, and not visiting them again.
- In a BFS or DFS traversal, edges visited form a tree (if the graph is connected) or a forest (if the graph is not connected).

### 5.1.2.2 Minimum Spanning Trees (MSTs)

- Use DFS to find a spanning tree in a graph, if one exists
  - Arbitrary tree
- Useful to find the “best” tree
  - Minimum Spanning Tree (e.g., minimum total length. Where length is: distance, cost, function(delay), function(reliability), etc.)

- If the graph is not connected → minimum spanning forest
  - For  $n$  nodes,  $c$  components, and  $e$  edges, we have:  $n = c + e$
  - For a tree,  $c = 1$ .
- DFS will not, in general, find the spanning tree with minimum total cost.

#### 5.1.2.2.1 *The Greedy Algorithm*

- At each stage, select the shortest edge possible.
- May not find a feasible solution when one exists.
- Efficient and simple to implement → widely used.
- Basis of other more complex and effective algorithms.
- In the case of MST, the greedy algorithm guarantees both optimality and reasonable computational complexity.
  - Start with empty solution  $s$
  - While elements exist
    - Find  $e$ , the best element not yet considered
    - If adding  $e$  to  $s$  is feasible, add it; if not, discard it.

#### 5.1.2.2.2 *Kruskal's Algorithm*

- A greedy algorithm for finding MSTs.
- Sort the edges, shortest first and then include all edges which do not form cycles with the edges previously selected.
- $n$ : number of nodes
- **Algorithm:**
  1. Sort all edges in ascending order (least cost first)
  2. Select among edges not yet selected, the one with the least cost.
  3. Add it if it does not create a cycle.
  4. If the number of edges selected  $< n-1$ , go to step (2), otherwise exit (tree completed)
- **Complexity:**

$O(m \log m)$ ,  $m$  = number of edges

### 5.1.2.2.3 Prim's Algorithm

- A greedy algorithm for finding MSTs.
- Advantageous if the network is dense.
- Well suited to parallel implementation.
- **Algorithm:**
  1. Start with one node (root node) in the tree
  2. Find node  $i$ , not in the tree, which is the nearest to the tree.
  3. Add node  $i$  to the tree and edge  $e$  connecting  $i$  to the tree.
- **Complexity:**

$$O(n^2)$$

### 5.1.2.2.4 Comparison of the Complexity of Kruskal's and Prim's Algorithms

- If the network is dense  $\rightarrow m \sim O(n^2) \rightarrow$  Prim's algorithm is faster
- If the network is not dense  $\rightarrow m \sim O(n) \rightarrow$  Kruskal' algorithm is faster

## 5.1.3 Constrained/Capacitated MST (CMST)

- The algorithms presented in the previous subsections are called “unconstrained MST algorithms”
  - No constraint on flow of information
  - No constraint on the number of ports at each node.
- For the unconstrained spanning tree problem, all these algorithms produce a minimum cost spanning tree.
- **CMST Problem:** Given a central node  $N_0$  and a set of other nodes  $(N_1, N_2, \dots, N_n)$ , as et of weights  $(W_1, W_2, \dots, W_n)$  for each node, the capacity of a link,  $W_{\max}$ , and a cost matrix  $C_{ij} = \text{Cost}(i,j)$ , find a set of trees  $T_1, T_2, \dots, T_k$  such that each  $N_i$  belongs to exactly one  $T_j$  and each  $T_j$  contains  $N_0$ .

- **Objective:** Find a tree of minimum cost and which satisfies a number of constraints such as:
  - Flow over a link
  - Number of ports
  
- **Example:**
  - Assume we are allowed to use one type of links only that can accommodate a maximum of 5 units of flow per unit time.
  - Assume that the flow generated from each node to the central node ( $N_1$ ) is as follows:  $f_1=0$ ,  $f_2=2$ ,  $f_3=3$ ,  $f_4=2$ ,  $f_5=1$  (in units/time\_unit).
  
- Effect of constraint violation:
  - As a result, a queue will build up since node 3 can service only 5 units/time\_unit. If node 3 does not have a large queue to accommodate all coming units, some units will be lost. So, these units are retransmitted, which may cause the network to collapse.
  
- The CSMT problem is NP-hard (i.e., cannot be solved in polynomial time)
  - Resort to heuristics (approximate algorithms)
  
- These heuristics will attempt to find a good feasible solution, not necessarily the best, that:
  - Minimizes the cost
  - Satisfies all the constraints
  
- Well-known heuristics:
  - Kruskal
  - Prim
  - Esau-Williams

### 5.1.3.1 Kruskal's Algorithm for CMST

#### Algorithm:

1. Sort all edges in ascending order,  $e \leftarrow 0$ .
2. Select edge with minimum cost (from edges not yet selected)
3. If it satisfies constraints (i.e., no cycles and no violation of flows on links)
  - o Then: add it to the tree,  $e \leftarrow e + 1$
  - o Else: go to step (2)
4. If ( $e = n - 1$ ) then exit, else go to step (2)

#### Example:

Given a network with five nodes, labelled **1** to **5**, and characterized by the following cost matrix:

	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>
<b>1</b>	-	3	3	5	10
<b>2</b>	3	-	6	4	8
<b>3</b>	3	6	-	3	5
<b>4</b>	5	4	3	-	7
<b>5</b>	10	8	5	7	-

Node 1 is the central backbone node.

$$f_{\max}=5, f_1=0, f_2=2, f_3=3, f_4=2, f_5=1.$$

### 5.1.3.2 Prim's Algorithm for CMST

#### Algorithm:

1. Start with one node (root node) in the tree.
2. Find node  $i$ , not in the tree, which is the nearest to the tree
3. Add node  $i$  to the tree and edge  $e$  connecting  $i$  to the tree if it satisfies constraints (i.e., no violation of flows on links)

#### Example:

### 5.1.3.3 Esau-Williams Algorithm for CMST

Node 1 is the central node.

$t_{ij}$ : is the tradeoff of connecting  $i$  to  $j$  or  $i$  directly to the root.

- If ( $t_{ij} < 0$ ) → better to connect  $i$  to  $j$
- If ( $t_{ij} \geq 0$ ) → better to connect  $i$  directly to the root

#### Algorithm:

1. Compute  $t_{ij} = c_{ij} - c_{i1}$  for all  $i, j \neq 1$ , and  $i \neq j$
2. Select the link  $(m,n)$  such that:  $t_{mn} = \min(t_{ij})$
3. If  $t_{mn} < 0$ , then go to step (4)  
Else (i.e.,  $t_{mn} \geq 0$  for all  $m,n$ ), connect to node 1 all nodes not connected yet, and **exit**.
4. Verify constraints (e.g., does not exceed the maximum weight)
  - If satisfied go to step (5)
  - Else:  $t_{mn} = \infty$  and  $t_{nm} = \infty$ , go to step (2)
5. Add link  $(m, n)$ , remove link  $(m, 1)$  and update  $t_{ij}$  to indicate that **m** is now connected to **n**.
  - $t_{mn} = \infty$  and  $t_{nm} = \infty$
  - if  $t_{mj} \neq \infty$ ,  $t_{mj} = c_{mj} - \min(c_{ki})$  [ $k \in C_m$ , where  $C_i$  = component containing node **i**]
6. Go to step (2)

#### Example:

Given a network with five nodes, labelled **1** to **5**, and characterized by the following cost matrix:

	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>
<b>1</b>	-	3	3	5	10
<b>2</b>	3	-	6	4	8
<b>3</b>	3	6	-	3	5
<b>4</b>	5	4	3	-	7
<b>5</b>	10	8	5	7	-

$W_{\max}=5, W_1=0, W_2=2, W_3=3, W_4=2, W_5=1.$





## 5.1.4 Terminal Assignment

### 5.1.4.1 Problem Statement

- **Terminal Assignment:** Association of terminals with specific concentrators.

#### Given:

T terminals (stations)  $i = 1, 2, \dots, T$

C Concentrators (hubs/switches)  $j = 1, 2, \dots, C$

$C_{ij}$ : cost of connecting terminal  $i$  to concentrator  $j$

$W_j$ : capacity of concentrator  $j$

Assume that terminal  $i$  requires  $W_i$  units of a concentrator capacity.

Assume that the cost of all concentrators is the same.

- $x_{ij} = 1$ ; if terminal  $i$  is assigned to concentrator  $j$ .

- $x_{ij} = 0$ ; otherwise.

#### Objective:

### 5.1.4.2 Augmenting path algorithm

#### Based on the following observations:

1. Ideally, every terminal is assigned to the nearest concentrator.
2. Terminals on concentrators that are full are moved only to make room for another terminal that would cause a higher overall cost if assigned to another concentrator.
3. An optimal partial solution with  $k+1$  terminals can be found by finding the least expensive way of adding the  $(k+1)^{\text{th}}$  terminal to the  $k$  terminal solution.

#### Assignment problem:

Given a cost matrix:

- One column per concentrator
- One row per terminal

Assume that:

- Weight of each terminal is 1 (i.e., each terminal consumes exactly one unit of concentrator capacity)
- A concentrator has a capacity of  $W$  terminals (e.g., number of ports)

A feasible solution exists iff  $T \leq W * C$



#### Algorithm:


1. Initially, try to associate each terminal to its nearest concentrator
2. If successful in assigning all terminals without violating capacity constraints, then stop (i.e., an optimal solution is found)
3. Else,
  - **Repeat**
    - i. Build a compressed auxiliary graph
    - ii. Find an optimal augmentation
  - **Until** all terminals are assigned

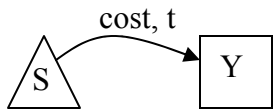
**Building a compressed auxiliary graph:**

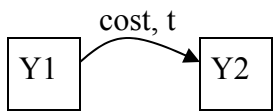
**U:** set of unassociated terminals

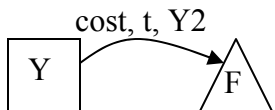
**T(Y):** set of terminals associated with Y

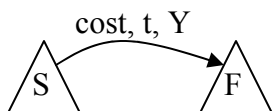
 and  are the start and finish of all augmenting paths

 represents a fully loaded concentrator

 **Assign t to a fully loaded concentrator Y. ( $t \in U$ )**  
 $\text{cost} = c(tY) = \min c(xY)$  for  $x \in U$

 **Move t from a fully loaded concentrator Y1 to another fully loaded concentrator Y2. ( $t \in T(Y1)$ )**  
 $\text{cost} = c(tY2) - c(tY1) = \min (c(xY2) - c(xY1))$  for  $x \in t(Y1)$

 **Move t from Y to a concentrator Y2 with spare capacity. ( $t \in T(Y)$ )**  
 $\text{cost} = c(tY2) - c(tY) = \min (c(xY2) - c(xY))$  for  $x \in t(Y)$

 **Assign t to a concentrator Y with spare capacity. ( $t \in U$ )**  
 $\text{cost} = c(tY) = \min c(xY)$  for  $x \in U$

**Example:**

## 5.2 Traffic Flow Analysis and Performance Evaluation

### 5.2.1 Traffic Flow Analysis Objective

- Estimate:
  - Delay
  - Utilization of resources (links)
- Traffic flow across a network depends on:
  - Topology
  - Routing
  - Traffic workload (from all traffic sources)
- Desirable topology and routing are associated with:
  - Low delays
  - Reasonable link utilization (no bottlenecks)
- Assumptions:
  - Topology is fixed and stable
  - Links and routers are 100% reliable
  - Processing time at the routers is negligible
  - Capacity of all links is given  $C = [C_i]$  (in bps [bits per second])
  - Traffic workload is given  $\Gamma = [\gamma_{jk}]$  (in pps [packets per second])
  - Routing is given  $R = [r_{jk}]$
  - Average packet size is  $1/\mu$  bits.

### 5.2.2 Queuing Analysis

Projections of performance are made on the basis of either:

- The existing load information, or
- The estimated load for the new environment.

Approaches that could be used:

- Do an after-the-fact analysis based on actual values
- Make a simple projection from existing to expected environment
- Develop an analytic model based on queuing theory
- Program and run a simulation tool

### 5.2.2.1 Queuing Models

- The notation **X/Y/N** is used for queuing models.
  - X = distribution of the interarrival times
  - Y = distribution of service times
  - N = number of servers
- The most common distributions are:
  - G = general independent arrivals or service times
  - M = negative exponential distribution
  - D = deterministic arrivals or fixed length service

➤ Example: **M/M/1**

➤ **Single-server queues**

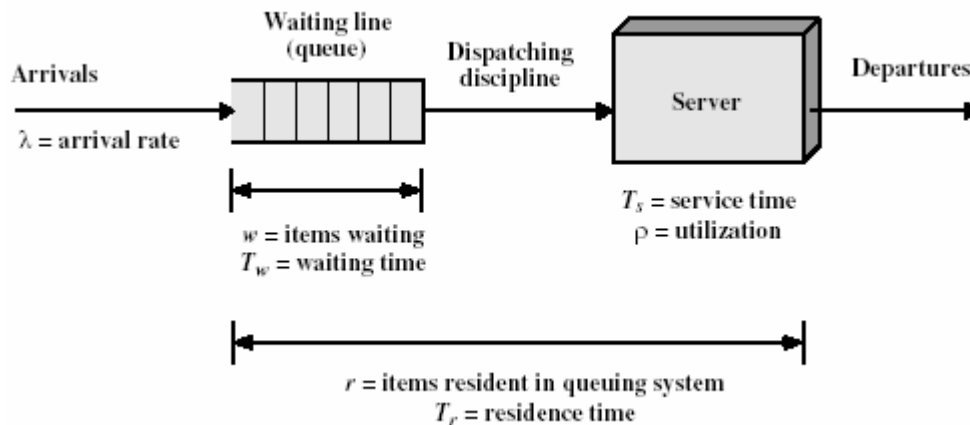


Figure 5.1: Queuing System Structure and Parameters for Single-Server Queue  
(Taken from "Queuing Analysis" by William Stallings)

#### Queue parameters:

$\lambda$  = arrival rate; mean number of arrivals per second

$T_s$  = mean service time for each arrival; amount of time being served, not counting time waiting in the queue

$\rho$  = utilization; fraction of time facility (server or servers) is busy

$r$  = mean number of items in system, waiting and being served (residence time)

$T_r$  = mean time an item spends in system (residence time)

$w$  = mean number of items waiting to be served

$T_w$  = mean waiting time (including items that have to wait and items with waiting time = 0)

### Basic Queuing relationship:

- $\rho = \lambda * T_s$
- $r = w + \rho$
- $\lambda_{\max} = 1/T_s$
- $r = \lambda * T_r$  (**Little's formula**)
- $w = \lambda * T_w$  (Little's formula)
- $T_r = T_w + T_s$
- $r = \rho / (1 - \rho)$

#### ➤ **Multiserver queue**

$N$  = number of servers

$\rho$  = utilization of each server

$N\rho$  = utilization of all servers ( $= \lambda * T_s$ )

### **5.2.2.2 M/M/1 Queues – Application to Networks**

- Each link is seen as a service station servicing packets.

$\lambda_i$  = arrival rate (in pps); mean number of packets that arrive to link  $i$  in one second.

$\mu C_i$  = average service rate (in pps); mean number of packets that will get out of the link  $i$  in one second. ( $= 1/T_s$ )

- Utilization of link  $i$  is:

$$\rho_i =$$



- Stability condition of a network is:

- The external workload offered to the network is:

$$\gamma =$$

Where:

$\gamma$  = total workload in packets per second

$\gamma_{jk}$  = workload between source  $j$  and destination  $k$

$N$  = total number of sources and destinations

- The internal workload on link  $i$  is:

$$\lambda_i =$$

Where:

$\gamma_{jk}$  = workload between source  $j$  and destination  $k$

$\Pi_{jk}$  = path followed by packets to go from source  $j$  and destination  $k$

- The total internal workload is:

$$\lambda =$$

Where:

$\lambda$  = total load on all of the links in the network

$\lambda_i$  = load on link  $i$

$L$  = total number of links

- The average length for all paths is given by:

$$\mathbf{E[\text{number of links in a path}] = \lambda/\gamma}$$

- The average number of items waiting and being served for link  $i$  is:

$$\mathbf{r_i =}$$

- The number of packets waiting and being served in the network can be expressed as:

$$\gamma * T =$$

Where:

T = average delay experienced by a packet through the network.

$$T =$$

- $T_{ri}$  is the residence time at each queue. If we assume that each queue can be treated as an independent M/M/1 model (Jackson's Theorem), then:

$$T_{ri} =$$

Where:  $T_{si}$  is the service time for link i

$$T_{si} =$$

Where:

- $C_i$  = data rate on the link (in bps)
- $M = 1/\mu$  = average packet length in bits

**Example:**



## 5.3 Network Reliability

### 5.3.1 Introduction

- A network model is a set of facilities. A facility could be a device or a link.
- A network must contain some slack to allow it to function even if some of its facilities have failed.
- Any network facility is either:
  - Working (**p**)
  - Failing ( $q = 1-p$ )
- **MTBF**: Mean Time Between Failures (**f**).
- **MTTR**: Mean Time To Repair (**r**)

- For any facility  $i$ , we'll know from measurements of  $f_i$  and  $r_i$ :

$$P_i = \text{Prob} [\text{facility } i \text{ is working}] =$$

Therefore:

- We assume that all facilities are independent:

$$P(\mathbf{ij}) = \text{Prob}[\text{facility } i \text{ and facility } j \text{ are working}] =$$

$$P(\mathbf{i|j}) = \text{Prob}[\text{facility } i \text{ or facility } j \text{ is working}] =$$

- Simplest measure of network reliability:

$$P_c(\mathbf{G}) = \text{Prob}[\text{Network is connected}]$$

Where: **c** stands for the connectivity of the network, and  
**G** stands for the graph representing the network

**$P_c(G) = \text{Prob}[\text{All nodes are working and there is a spanning tree of working links}]$**

**$P_c(G) =$**

- Since enumerating all trees in  $G$  requires an exponential amount of effort,  $P_c(G)$  is very difficult (if not impossible) to compute.

→ We seek simpler measures of network reliability.

### 5.3.2 Reliability of Tree Networks

- A typical enterprise/campus network includes trees:

- Given a tree  $T$ :

**$P_c(T) = \text{Prob}[\text{A tree network, } T, \text{ being connected}]$   
 **$= \text{Prob}[\text{All components (nodes and links) are working}]$****

**$P_c(T) =$**

- $P_c(T)$  can also be computed recursively:

**$P_c(T) =$**

Where:  **$T-i$**  is the tree  $T$  without node  $i$ , and  
 **$j$**  is the link between node  $i$  and the rest of the tree

- Given a particular tree with root r:

$$P_c(\mathbf{i}) = \text{Prob}[\text{node } i \text{ can communicate with root } r]$$

$$P_c(\mathbf{i}) =$$

Where:  $\mathbf{j}$  is the link between nodes  $i$  and  $k$ , and  
 $\mathbf{k}$  is the predecessor of node  $i$

$$P_c(\mathbf{r}) =$$

- The expected number of nodes communicating with the root r is:

$$E(\mathbf{r}) =$$

- This expression can be computed efficiently for any node as follows:

$E(\mathbf{i})$  = the expected number of nodes communicating with the node  $i$

$$E(\mathbf{i}) =$$

- If node  $i$  is a leaf, then:

$$E(\mathbf{i}) =$$

**Example:**



- The expected number of node pairs communicating through the root  $r$  is:

$$\mathbf{EPR(r) =}$$

**Example:**

## **5.4 References**

1. “Telecommunications Network Design Algorithms” by Aaron Kershenbaum, 1993
2. “Queuing Analysis” by William Stalling, 2000