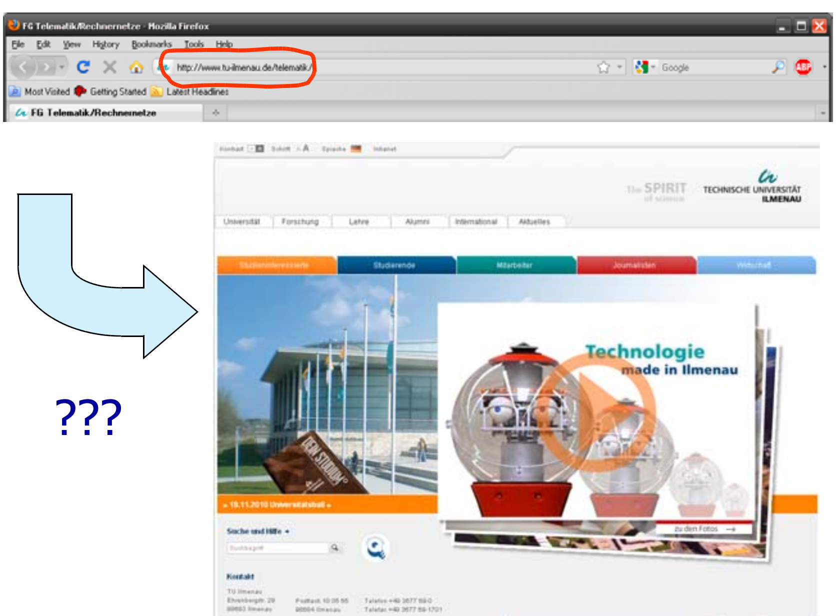
Algorithmic Aspects of Communication Networks

Chapter 1
Introduction

- Short Recapitulation of Networking Basics
- Packet Forwarding and Routing
- Characterizing Traffic
- Traffic Demand and Link Utilization

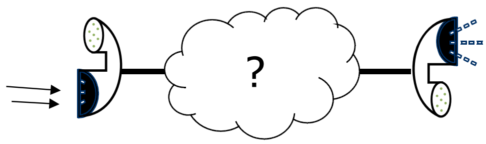
Short Recap: World Wide Web

- You already know from a basic networking class what happens when you enter *http://www.tu-ilmenau.de* into a Web browser



Short Recap: Telephony

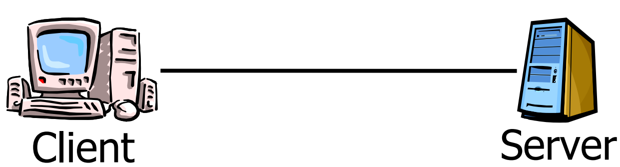
- Also, you have a basic understanding what happens when picking up a telephone and making a phone call
  - How to find the peer's phone? How to transmit speech?



- You are aware of the differences between transferring a Web page and a phone call (and their implications):
  - Web: Bunch of data that has to be transmitted
  - Phone: *Continuous* flow of information, must arrive *in time*

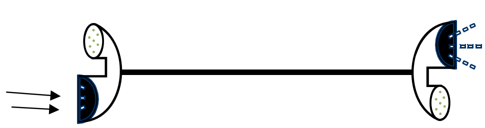
Simplest Communication: Direct Physical Connection

- Web example: Browser = client and server
  - Simplest case: directly connect them by a (pair of) cable



  - Server provides data, client consumes it

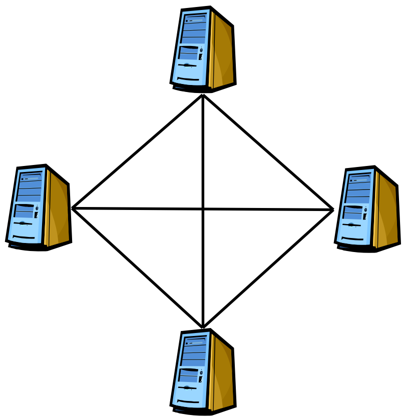
- Telephony: Connect two telephones via a (pair of) cable



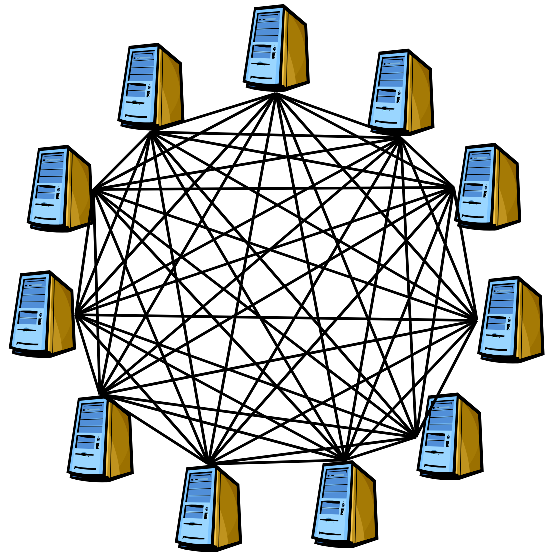
But There are More than Two Computers / Telephones

- Connect each telephone / computer with each other one?

With four computers:



With eleven computers:



...

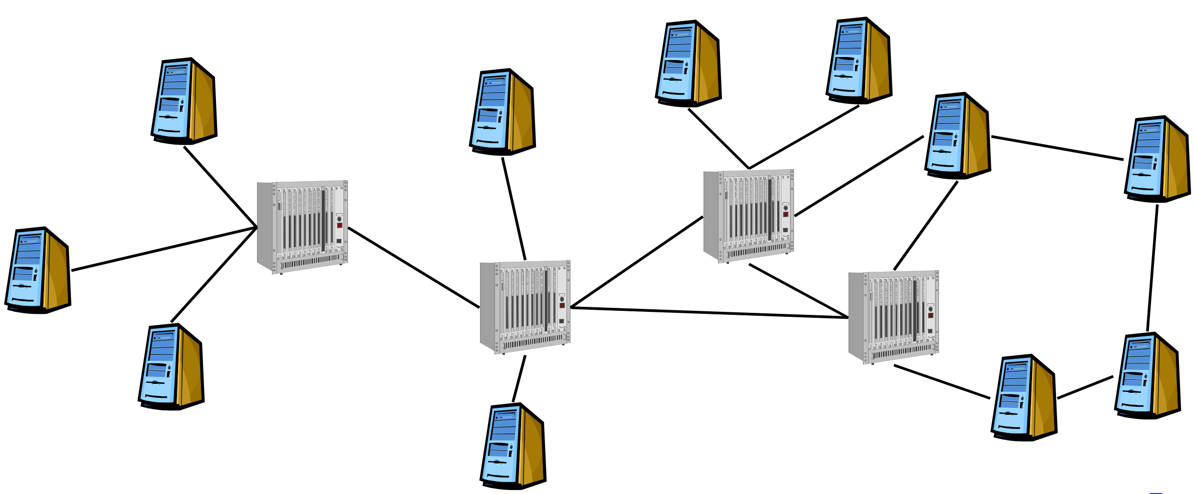
Beirut Connections

- Connecting many phones in real life

Beirut Connections

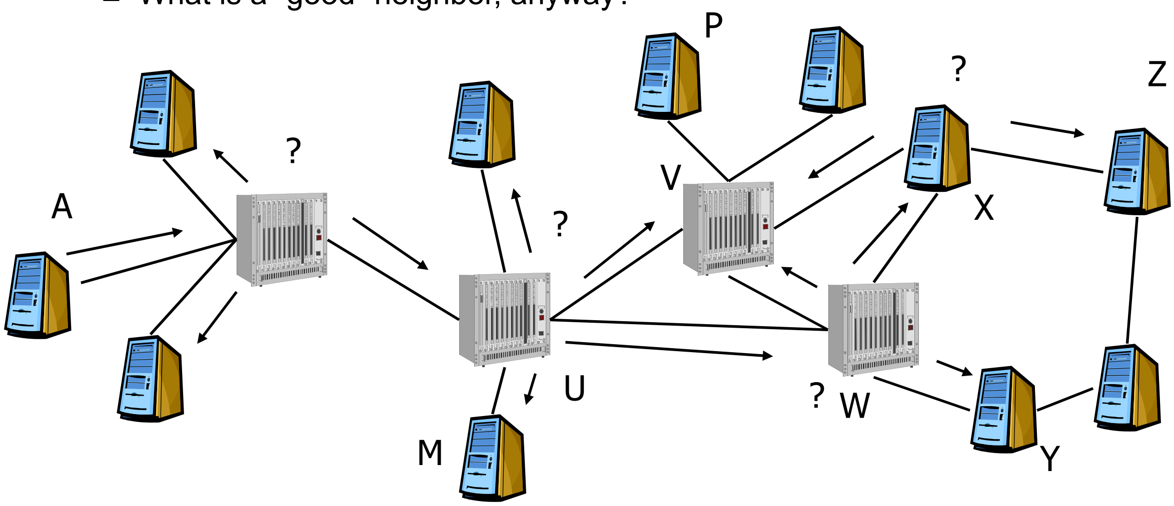
Put some Structure into a Network

- Pair wise connecting all entities does not work
- Need some structure
  - Distinguish between "*end systems/terminals/user devices*" on one hand, "*switching elements/routers*" on the other hand

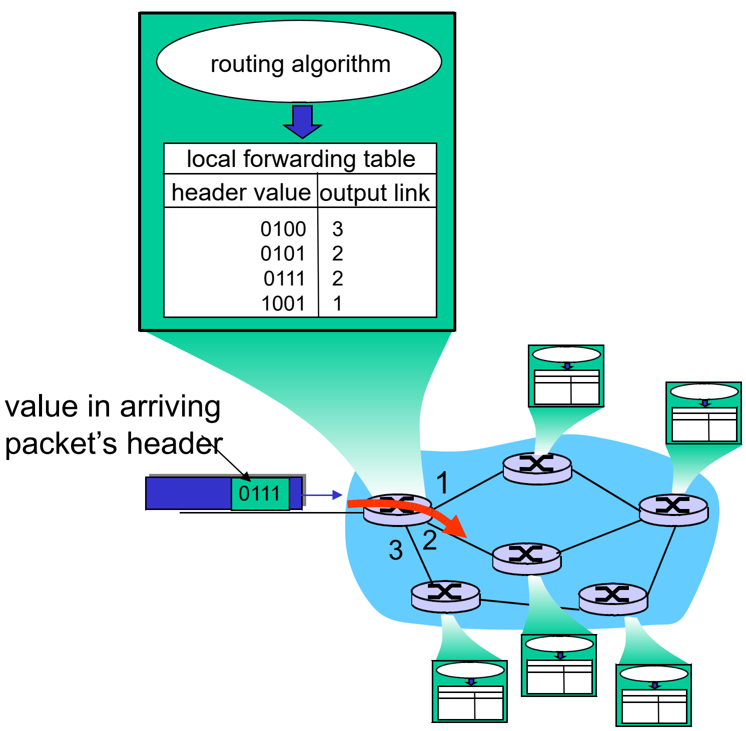


Forwarding and Next Hop Selection

- Recall: A switching element/a router *forwards* a packet onto the next hop towards its destination
- How does a router *know* which of its neighbors is the best possible one towards a given destination?
  - What is a “good” neighbor, anyway?



Interplay Between Routing and Forwarding



Overview on Routing Algorithms (1)

- An router executes a routing algorithm to decide which output line an incoming packet should be transmitted on:
  - In connection-oriented service, the routing algorithm is performed only during connection setup
  - In connectionless service, the routing algorithm is either performed as each packet arrives, or performed periodically and the results of this execution updated in the forwarding table
- Often, routing algorithms take a so-called *metric* into account when making routing decisions:
  - In this context, a metric assigns a cost to each network link
  - This allows to compute a metric for each route in the network
  - A metric may take into account parameters like number of hops, "€ cost" of a link, delay, length of output queue, etc.
  - The "cheapest" path according to some metric is often also referred to as the *shortest path* (even though it might actually contain more hops than an alternative path)

Overview on Routing Algorithms (2)

- Two basic types of routing algorithms:
  - *Non-adaptive routing algorithms:* do not base their routing decisions on the current state of the network (example: flooding)
  - *Adaptive routing algorithms:* take into account the current network state when making routing decisions (examples: distance vector routing, link state routing)
- Remark: additionally, hierarchical routing can be used to make these algorithms scale to large networks

Flooding

- Basic strategy:
  - Every incoming packet is sent out on every outgoing line except the one it arrived on
  - Problem: vast number of duplicated packets
- Reducing the number of duplicated packets:
  - Solution 1:
    - Have a hop counter in the packet header; routers decrement each arriving packet's hop counter; routers discard a packet with hop count=0
    - Ideally, the hop counter should be initialized to the length of the path from the source to the destination
  - Solution 2:
    - Require the first router hop to put a sequence number in each packet it receives from its hosts
    - Each router maintains a table listing the sequence numbers it has seen from each first-hop router; the router can then discard packets it has already seen

Adaptive Routing Algorithms (1)

- Problems with non-adaptive algorithms:
  - If traffic levels in different parts of the subnet change dramatically and often, non-adaptive routing algorithms are unable to cope with these changes
  - Lots of computer traffic is *bursty* (~ very variable in intensity), but non-adaptive routing algorithms are usually based on average traffic conditions
  - → Adaptive routing algorithms can deal with these situations
- Three types:
  - *Centralized adaptive routing:*
    - One central routing controller
  - *Isolated adaptive routing:*
    - Based on local information
    - Does not require exchange of information between routers
  - *Distributed adaptive routing:*
    - Routers periodically exchange information and compute updated routing information to be stored in their forwarding table

Centralized Adaptive Routing

- Basic strategy:
  - Routing table adapts to network traffic
  - A routing control center is somewhere in the network
  - Periodically, each router forwards link status information to the control center
  - The center can compute the best routes, e.g. with Dijkstra's shortest path algorithm (explained later)
  - Best routes are dispatched to each router
- Problems:
  - Vulnerability: if the control center goes down, routing becomes non-adaptive
  - Scalability: the control center must handle a great deal of routing information, especially for larger networks

Isolated Adaptive Routing Algorithms

- Basic idea:
  - Routing decisions are made only on the basis of information available locally in each router
- Examples:
  - Hot potato
  - Backward learning
- Hot potato routing:
  - When a packet arrives, the router tries to get rid of it as fast as it can by putting it on the output line that has the shortest queue
  - Hot potato does not care where the output line leads
  - Not very effective

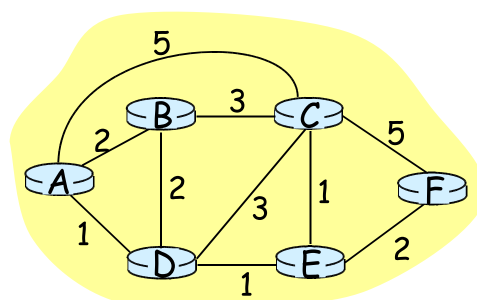
Backward Learning Routing

- Basic idea:
  - Packet headers include destination and source addresses; they also include a hop counter
    → learn from this data as packets pass by
  - Network nodes, initially ignorant of network topology, acquire knowledge of the network state as packets are handled
- Algorithm:
  - Routing is originally random (or hot potato, or flooding)
  - A packet with a hop count of one is from a directly connected node; thus, neighboring nodes are identified with their connecting links
  - A packet with a hop count of two is from a source two hops away, etc.
  - As packets arrive, the IMP compares the hop count for a given source address with the minimum hop count already registered; if the new one is less, it is substituted for the previous one
  - Remark: in order to be able to adapt to deterioration of routes (e.g. link failures) the acquired information has to be "forgotten" periodically

Distributed Adaptive Routing

Routing Protocol

Goal: Determine "good" path (sequence of routers) through network from source to dest.



Graph abstraction for routing algorithms:
- Graph nodes are routers
- Graph edges are physical links
  - Link cost: delay, $ cost, or congestion level
  - Path cost: sum of the link costs on the path

- "Good" path:
  - Typically means minimum cost path
  - Other definitions possible

Decentralized Adaptive Routing Algorithm Classification

Global or decentralized information?

Decentralized:

- Router knows physically-connected neighbors, link costs to neighbors
- Iterative process of computation, exchange of info with neighbors
- "Distance vector" algorithms
  - → RIP protocol
  - → BGP protocol ("path vector")

Global:

- All routers have complete topology, link cost info
- "Link state" algorithms
  - → Dijkstra's algorithm
  - → OSPF protocol

Static or dynamic?

Static:

- Routes change slowly over time

Dynamic:

- Routes change more quickly
  - Periodic update
  - In response to link cost changes

Graph Model for Routing Algorithms (1)

- Input: a graph G=(V, E) with
  - V = $\{v_1, v_2, \ldots, v_n\}$ the set of nodes
  - $E \subseteq V \times V$; $E=\{e_1, e_2, \ldots, e_m\}$ the set of edges
  - A mapping $c(v_i, v_j)$ representing the cost of edge (v_i, v_j) if (v_i, v_j) is in E (otherwise $c(v_i, v_j)$ = infinity)
  - Start node $s=v_x$ an arbitrary node from the set V
- Output: two arrays d and p with
  - d[i] containing the cost (distance) of the shortest path from s to v_i
  - p[i] containing the index j of the predecessor node v_j of v_i on the shortest path from s to v_i

Graph Model for Routing Algorithms (2)

- Assumptions and Notation:
  - Let $s \in V$ be the source node
  - $\forall v \in V$: let $\delta(s, v)$ denote the cost of the shortest path from s to v
  - While d[i] denotes the shortest path cost (estimate) for node v_i, we also write this as $d(v_i)$ in our proof later on
  - Assume that source node s is connected to every node v in the graph:
    - $\forall s, v \in V: \delta(s, v) < \infty$
  - Assume all edge weights are finite and positive:
    - $\forall (v, w) \in E: c(v, w) > 0$

Dijkstra's Algorithm for Shortest Paths (1)

- Let us try to develop an algorithm for computing shortest paths by induction:
  - We could try induction on nodes or on edges: we choose nodes here
  - We need an estimate of costs to all nodes
    - Initially, this is set to infinity for all nodes except the source node s: $\forall v_i \in V \setminus \{s\}$: d[i] := ∞
    - Cost estimate for node $s=v_x$ is set to 0: d[x] := 0
  - When trying to "reduce" our problem to a smaller one by making use of induction, it would not be wise to actually remove nodes from the graph, as this would change the graph (e.g. destroy connectivity)
  - Therefore, we run our induction on the number *n* of nodes for which we can compute the shortest paths
  - <u>Base case:</u> *n* = 1: We know how to compute the shortest path from s to s
  - <u>Induction hypothesis:</u> We know how to compute the lengths of the shortest paths and respective predecessor nodes for up to *n* nodes.

Dijkstra's Algorithm for Shortest Paths (2)

- Induction step: $n \rightsquigarrow n + 1$
- Consider the set N of nodes for which we know the lengths of their shortest paths as well as the predecessor nodes:
  - Initially, N := {s}
  - So, we would like to increase the set N in every step by one node
  - But, which node of V \ N can be selected?
  - Clearly, it is not a good idea to choose a node v_i for which our current shortest path cost estimate is high, e.g. d[i] = ∞
- How to get better estimates?
  - Whenever we insert a node v_i into N (also when s is inserted to N), we can update our estimates for nodes v_j that are adjacent to v_i:

```
if ( d[i]+ c[i, j] < d[j]) {
   d[j] = d[i] + c[i, j];
   p[j] = i; }
```

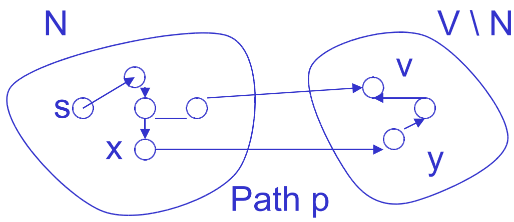
  - As d[i] is the cost of a path from s to v_i, the cost of the shortest path from s to v_j can not be higher than d[i]+c[i, j]

Dijkstra's Algorithm for Shortest Paths (3)

- Actually, we will show now by contradiction that in every step the vertex v_i in V \ N with a minimal value *d[i]* can be inserted into N and that d[i] equals the shortest path cost from source *s* to v_i (and at all subsequent times)
  - With this it is trivial to see that also the predecessor node is correctly set
- Too see this, let us assume that this is not true when the *n+1*th node v is added to *N*
  - Thus the vertex v added has $d(v) > \delta(s, v)$
  - Consider the situation just before insertion of v
  - Consider the true shortest path *p* from *s* to *v* (see next slide)

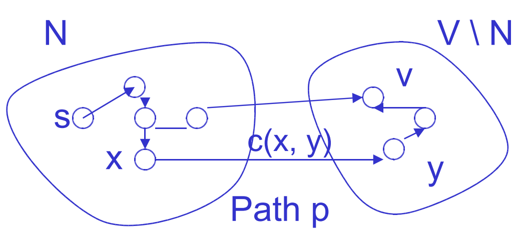
Dijkstra's Algorithm for Shortest Paths (4)

- Since s is in N, and v is in $V \setminus N$, path p must jump from N to $V \setminus N$ at some point:



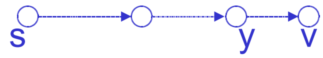
- Let the jump have end point x in N, and y in $V \setminus N$ (possibly $s = x$, and / or $v = y$)
- We will argue that y and v are different nodes
- Since path p is the shortest path from s to v, the segment of path p between s and x, is the shortest path from s to x, and that between s and y is the shortest path from s to y

Dijkstra's Algorithm for Shortest Paths (5)



- The cost of the segment between s and x is $d(x)$, and since x is in N, $d(x)$ is the cost $\delta(s, x)$ of the shortest path to x (induction hypothesis)
- The cost of the segment between s and y is $d(x) + c(x, y)$
  - Thus, $\delta(s, y) = d(x) + c(x, y)$
- Because $d(y)$ was correctly updated when node x was inserted into N it also holds that $d(y) \leq d(x) + c(x, y) = \delta(s, y)$
- This implies that $d(y) = \delta(s, y)$
- However, as we assume that $d(v) > \delta(s, v)$, v and y have to be different

Dijkstra's Algorithm for Shortest Paths (6)



- Since *y* appears somewhere along the shortest path between *s* and *v*, but *y* and *v* are different, we can deduce that $\delta(s, y) < \delta(s, v)$
  - Here we use the assumption that all edges have positive cost
- Hence, $d(y) = \delta(s, y) < \delta(s, v) < d(v)$
- As both *y* and *v* are in $V \setminus N$, *v* can not have been chosen in this step, as the algorithm always chooses the node *w* with minimum $d(w)$
- So, whenever a node *v* is included in *N*, it holds that $d(v) = \delta(s, v)$

∎

Dijkstra's Algorithm for Shortest Paths (7)

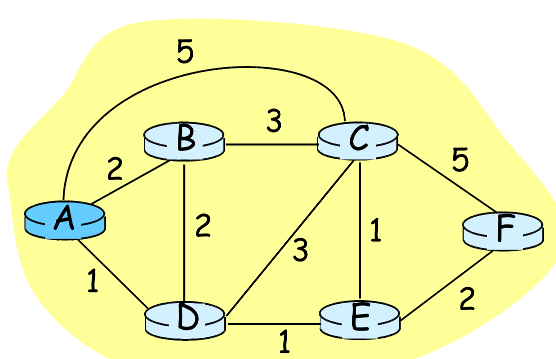
- Termination of Dijkstra's algorithm:
  - As the algorithm adds one node to *N* in every step, the algorithm terminates when the costs of shortest paths to all nodes have been computed correctly
- Algorithm complexity for $|V|$ nodes:
  - Each iteration: need to check all nodes *v* that are not yet in *N*
  - This requires $|V| \cdot (|V|+1)/2$ comparisons, leading to $O(|V|^2)$
  - This is optimal in dense graphs (where $|E| \sim |V|^2$)
  - In sparse graphs, more efficient implementations are possible: $O(|V| \cdot \log|V| + |E|)$ using so-called Fibonacci-heaps

Dijkstra's Algorithm in Pseudocode

```
Computes least-cost path from one node to other nodes in the net
Initialization (for Node s):
  N = {s} /* Set of Nodes with known least-cost path */
  for all nodes v
    if v adjacent to s
      then { d(v) = c(s,v); p(v) = s; }
      else d(v) = infinity;

Loop
   find v not in N such that d(v) is a minimum
   add v to N
   for all w adjacent to v and not in N do     /* update d(w) */
      { if (d(v) +c(v, w) < d(w))
           { d(w) = d(v) + c(v, w); p(w) = v; } }
   /* new cost to w is either old cost to w or known
      shortest path cost to v plus cost from v to w */
until all nodes in N
```

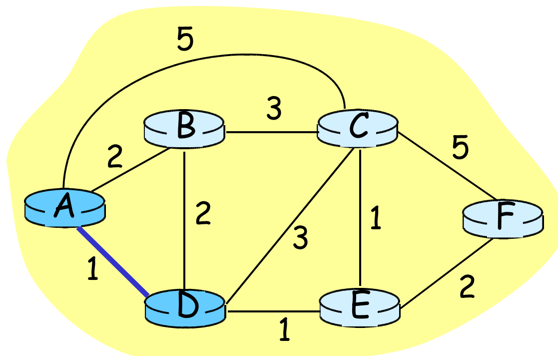

Example Run of Dijkstra's Algorithm (1)

Step	start N	d(B),p(B)	d(C),p(C)	d(D),p(D)	d(E),p(E)	d(F),p(F)
0	A	2,A	5,A	1,A	infinity	infinity



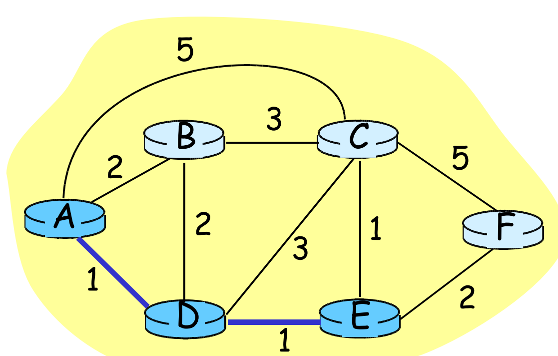
Example Run of Dijkstra's Algorithm (2)

Step	start N	d(B),p(B)	d(C),p(C)	d(D),p(D)	d(E),p(E)	d(F),p(F)
0	A	2,A	5,A	1,A	infinity	infinity
1	AD	2,A	4,D		2,D	infinity



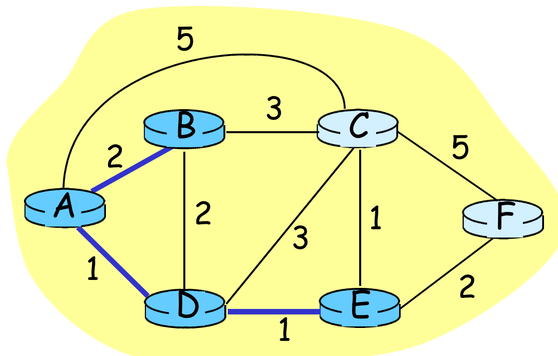
Example Run of Dijkstra's Algorithm (3)

Step	start N	d(B),p(B)	d(C),p(C)	d(D),p(D)	d(E),p(E)	d(F),p(F)
0	A	2,A	5,A	1,A	infinity	infinity
1	AD	2,A	4,D		2,D	infinity
2	ADE	2,A	3,E			4,E
3						
4						
5						



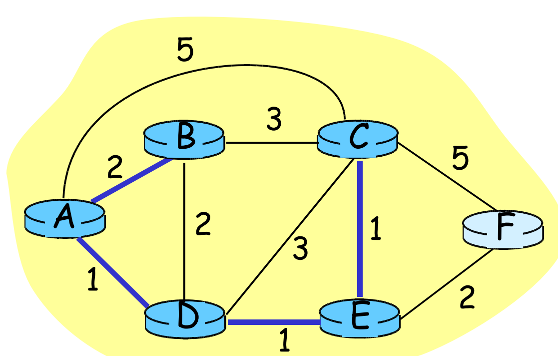
Example Run of Dijkstra's Algorithm (4)

Step	start N	d(B),p(B)	d(C),p(C)	d(D),p(D)	d(E),p(E)	d(F),p(F)
0	A	2,A	5,A	1,A	infinity	infinity
1	AD	2,A	4,D		2,D	infinity
2	ADE	2,A	3,E			4,E
3	ADEB		3,E			4,E



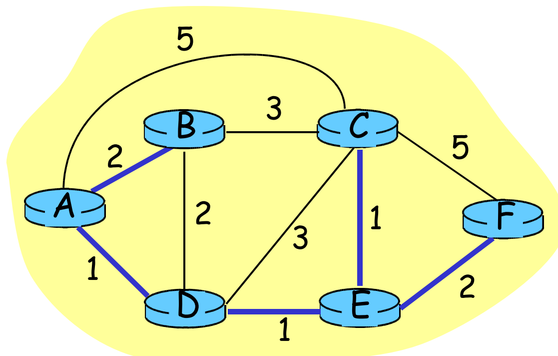
Example Run of Dijkstra's Algorithm (5)

Step	start N	d(B),p(B)	d(C),p(C)	d(D),p(D)	d(E),p(E)	d(F),p(F)
0	A	2,A	5,A	1,A	infinity	infinity
1	AD	2,A	4,D		2,D	infinity
2	ADE	2,A	3,E			4,E
3	ADEB		3,E			4,E
4	ADEBC					4,E



Example Run of Dijkstra's Algorithm (6)

Step	start N	d(B),p(B)	d(C),p(C)	d(D),p(D)	d(E),p(E)	d(F),p(F)
0	A	2,A	5,A	1,A	infinity	infinity
1	AD	2,A	4,D		2,D	infinity
2	ADE	2,A	3,E			4,E
3	ADEB		3,E			4,E
4	ADEBC					4,E
5	ADEBCF					



Distance Vector Routing Algorithm

Iterative:
- Continues until no nodes exchange info.
- *Self-terminating*: no "signal" to stop

Asynchronous:
- Nodes need *not* exchange info/iterate in lock step!

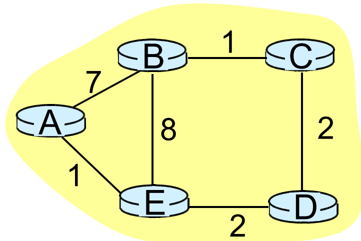
Distributed:
- Each node communicates *only* with directly-attached neighbors

Distance Table data structure
- Each node has its own
  - Row for each possible destination
  - Column for each directly-attached neighbor to node
- Example: in node X, for dest. Y via neighbor Z:

$D^X(Y,Z)$ = distance *from* X *to* Y, *via* Z as next hop

$= c(X,Z) + \min_w \{D^Z(Y,w)\}$

Example for Distance Table



$$D^E(C,D) = c(E,D) + \min_w \{D^D(C,w)\}$$
$$= 2+2 = 4$$
$$D^E(A,D) = c(E,D) + \min_w \{D^D(A,w)\}$$
$$= 2+3 = 5$$ loop!
$$D^E(A,B) = c(E,B) + \min_w \{D^B(A,w)\}$$
$$= 8+6 = 14$$ loop!

cost to destination via

$D^E()$ (destination)	A	B	D
A	1	14	5
B	7	8	5
C	6	9	4
D	4	11	2

Constructing Routing Table from Distance Table

Cost to destination via

$D^E()$ (Destination)	A	B	D
A	1	14	5
B	7	8	5
C	6	9	4
D	4	11	2

Destination	Outgoing link to use, cost
A	A,1
B	D,5
C	D,4
D	D,4

Distance table → Routing table

Distance Vector Routing: Overview

Iterative, asynchronous:

Each local iteration caused by:

- Local link cost change
- Message from neighbor: its least cost path change from neighbor

Distributed:

Each node notifies neighbors *only* when its least cost path to any destination changes

- Neighbors then notify their neighbors if necessary

Each node:

wait for (change in local link cost of msg from neighbor)

recompute distance table

if least cost path to any dest has changed, *notify* neighbors

Distance Vector Algorithm: Initialization

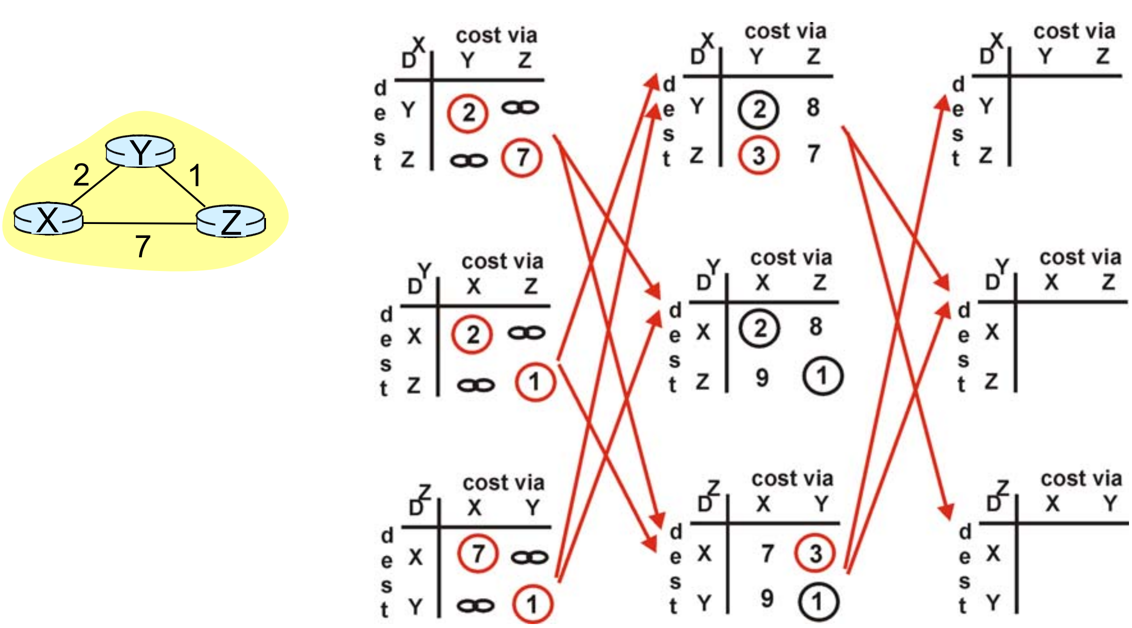
At all nodes, X:

```
Initialization:
  for all adjacent nodes v:
     D^X(*,v) = infinity       /* the * operator means "for all rows" */
     D^X(v,v) = c(X,v)
  for all destinations, y
     send min_w  D^X(y,w) to each neighbor  /* w over all X's neighbors */
```

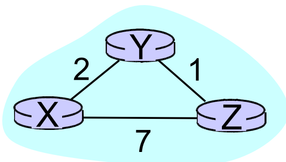
Distance Vector Algorithm: Main Loop

```
loop
  wait (until I see a link cost change to neighbor V
        or until I receive update from neighbor V)

   if (c(X,V) changes by d)
     /* change cost to all dest's via neighbor v by d */
     /* note: d could be positive or negative */
     for all destinations y:  D^X(y,V) =  D^X(y,V) + d

   else if (update received from V w.r.t. destination Y)
     /* shortest path from V to some Y has changed  */
     /* V has sent a new value for its  min_W  D^V(Y,w) */
     /* call this received new value "newval"     */
     for the single destination y: D^X(y,V) = c(X,V) + newval

   if we have a new min_W D^X(Y,w) for any destination Y
      send new value of min_W D^X(Y,w) to all neighbors

forever
```


Distance Vector Algorithm: Example (1)



Distance Vector Algorithm: Example (2)



D^X (dest)	cost via Y	cost via Z
Y	2	∞
Z	∞	7

D^Y (dest)	cost via X	cost via Z
X	2	∞
Z	∞	1

D^Z (dest)	cost via X	cost via Y
X	7	∞
Y	∞	1

D^X (dest)	cost via Y	cost via Z
Y	2	8
Z	3	7

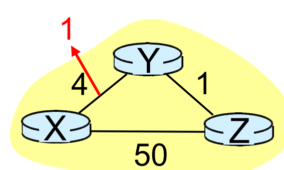
$$D^X(Y,Z) = c(X,Z) + \min_w\{D^Z(Y,w)\} = 7+1 = 8$$

$$D^X(Z,Y) = c(X,Y) + \min_w\{D^Y(Z,w)\} = 2+1 = 3$$

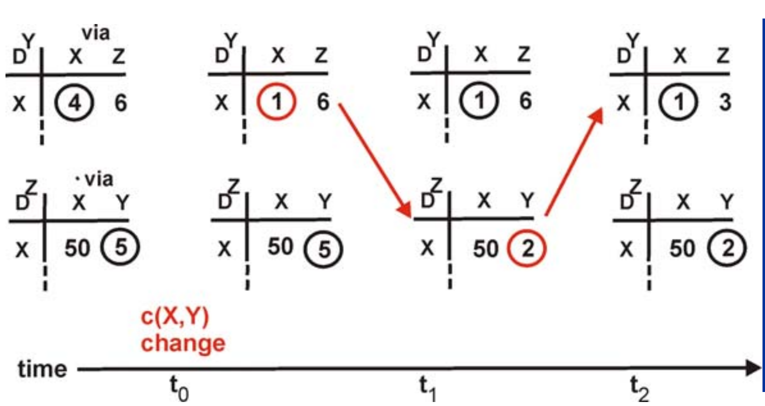
Distance Vector: Reaction to Link Cost Changes (1)

Link cost changes:

- Node detects local link cost change
- Updates distance table (line 15)
- If cost change in least cost path, notify neighbors (lines 23,24)



“Good news travels fast”

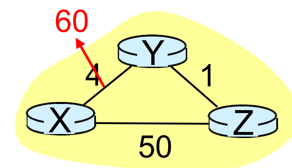


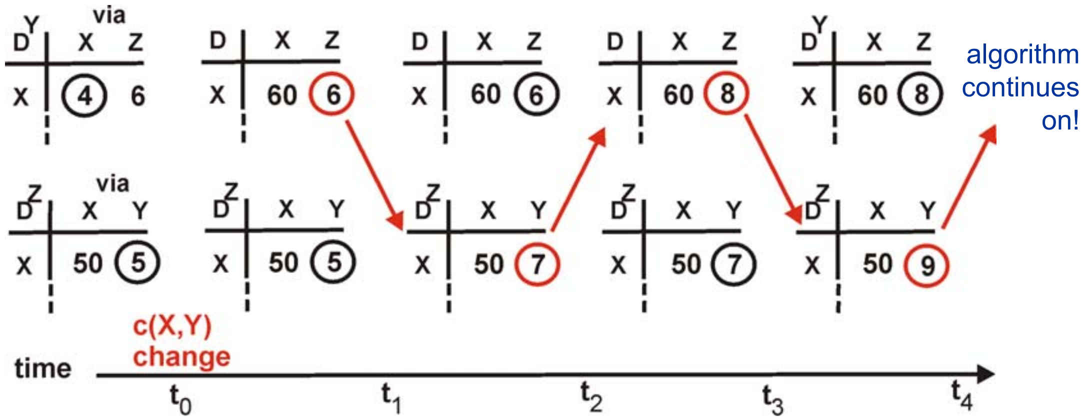
Algorithm “terminates”

Distance Vector: Reaction to Link Cost Changes (2)

Link cost changes:

- Good news travels fast
- Bad news travels slow - "count to infinity" problem!

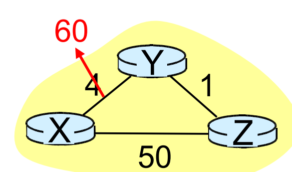


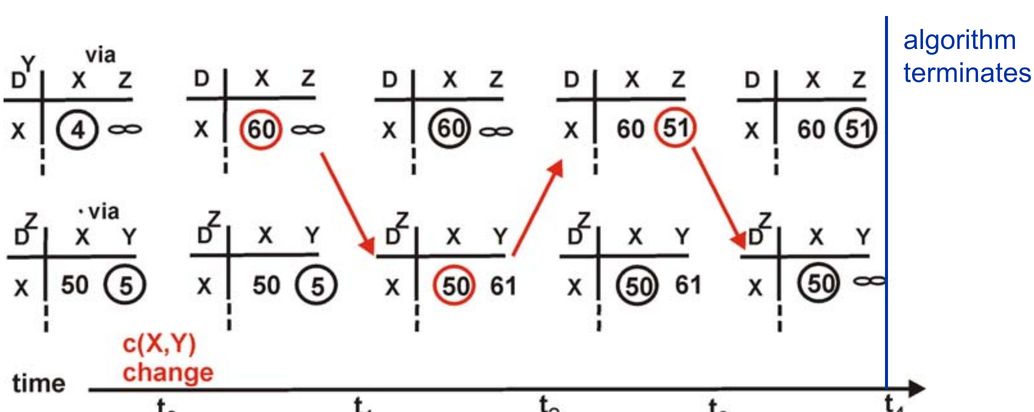


Distance Vector: Poisoned Reverse

If Z routes through Y to get to X:

- Z tells Y its (Z's) distance to X is infinite (so Y won't route to X via Z)
- Will this completely solve count to infinity problem?





The Bellman-Ford Algorithm (1)

- In our correctness proof of Dijkstra's algorithm, we assumed that all link costs are positive: $\forall (v, w) \in E: c(v, w) > 0$
  - In fact, it is possible to proof that the algorithm also correctly computes the shortest path from *s* to all nodes *v* in case that link costs may be zero (see [CLR90, section 25.2])
- The Bellman-Ford algorithm is capable of solving the even more general problem of computing shortest paths in graphs in which edges with negative cost exist:
  - In order for the shortest paths to be well defined in such graphs, it is required that there exist no negative weight cycles in the graph
  - Otherwise, the cost of the shortest path would "grow" to $-\infty$ on a path with infinite length containing infinite many tours along one or more negative weight cycles
  - When being run on a graph $G=(V, E)$ the algorithm detects, if there are negative weight cycles in *G* and computes the shortest paths from one source node *s* to all other nodes, in case that no such cycles exist

The Bellman-Ford Algorithm (2)

- Why do we care about this algorithm at all, and why at this point of the lecture?
  - First, because we have to care about this algorithm, as it is the basis of distance vector routing
  - Second, because it is easier to understand its proof after having learned about Dijkstra's algorithm
- Like Dijkstra's algorithm, the Bellman-Ford algorithm iteratively improves an estimate of the cost to reach each node:
  - The idea of the algorithm is to iterate $|V|$-1 times over all edges $(u, v) \in E$ and check, if the current estimate for the node *v* can be improved by making use of edge *(u, v)* given the current estimate of the cost to reach node *u*
  - Compared with Dijkstra's algorithm, which does not need to reconsider edges of nodes that have already been included in the set *N*, the Bellman-Ford algorithm always checks all edges, leading to a higher running time in the order of $O(|V| \cdot |E|)$

The Bellman-Ford Algorithm (3)

Computes least-cost path from one node to other nodes in the net

```
Initialization (for Node s):
  d(s) = 0;
  for all nodes v ≠ s { d(v) = infinity; p(v) = NULL; }
Main Algorithm
  for i = 1 to |V| - 1 {
    for all edges (u, v) ∈ E  { /* see if (u, v) can improve d(v) */
      if (d(u) +c(u, v) < d(v)) {
          d(v) = d(u) + c(u, v); p(v) = u;
      } /* of <if (u, v) can improve current d(v) */
    } /* of <for all edges>
 } /* of <for i> */
Check for negative cycles
for all edges (u, v) ∈ E  { /* see if (u, v) can still improve d(v) */
  if (d(u) +c(u, v) < d(v)) return FALSE;
}
return TRUE;
```


The Bellman-Ford Algorithm (4)

- Intuition behind the check for negative cycles:
  - In a graph with $|V|$ nodes and no negative cycles, a shortest path from node s to any node v, can at most have $|V|$-1 edges
  - We will see, that after the i-th iteration, all lengths of shortest paths to nodes v_i that are i hops away from s have been properly computed
  - Thus, after $|V|$-1 iterations, all shortest paths with a length of up to $|V|$-1 have been properly computed
  - So, if a further improvement is possible, this implies that the resulting path must have a length > $|V|$-1, and it can therefore be concluded that such a path must contain a negative cost cycle

Correctness of the Bellman-Ford Algorithm (1)

- Let $G=(V, E)$ be a graph with no negative-cost cycles reachable from a source node $s \in V$. Then, after termination of the Bellman-Ford algorithm it holds: $\forall\ v \in V$ *reachable from* s: $d(v) = \delta(s, v)$
- Proof:
  - Let v be a node reachable from s, and let $p=(v_0, v_1, ..., v_k)$ be a shortest path from s to v, where $v_0 = s$ and $v_k = v$
  - As G does not contain negative-cost cycles, the path p is simple and it holds that $k \leq |V| - 1$
  - We will prove by induction over i that after the i-th iteration over all edges of G it holds that $d(v_i) = \delta(s, v_i)$
  - Base case $i = 0$: $d(v_0) = \delta(s, v_0) = 0$
  - Inductive step from i to $i+1$:
    - By induction hypothesis we know that $d(v_{i-1}) = \delta(s, v_{i-1})$
    - As the edge (v_{i-1}, v_i) is checked in the i-th iteration to be part of the shortest path from s to v based on the current estimate for v_{i-1} (which is definite) we can conclude that $d(v_i) = \delta(s, v_i)$ ■

Correctness of the Bellman-Ford Algorithm (2)

- The Bellman-Ford algorithm when run over a weighted, directed graph $G=(V, E)$ with a source node s and a cost function c: E $\rightarrow$ |R returns
  - TRUE if there is no negative-cost cycle and it holds $\forall\ v \in V$: $d(v) = \delta(s, v)$
  - FALSE if there is a negative-cost cycle reachable from s
- Proof:
  - If the graph contains no negative-cost cycle reachable from s, then the result presented on the preceding slide proves the claim regarding $d(v)$
  - Furthermore, at the termination of Bellman-Ford we have:
    - $\forall\ (u, v) \in E$: $d(v) = \delta(s, v) \leq \delta(s, u) + c(u, v) = d(u) + c(u, v)$
    - This holds because the shortest path from s to v has no more weight than any other path from s to v, especially than the path which contains the edge (u, v)
  - Therefore, none of the tests in lines 13 to 15 returns FALSE and so the algorithm returns TRUE

Correctness of the Bellman-Ford Algorithm (3)

- Let us now assume that G contains a negative-cost cycle reachable from s:
  - Let z = $(v_0, v_1, \ldots, v_k)$ be the negative-cost cycle with $v_0 = v_k$
  - Thus it holds that $\sum_{i=1}^{k} c(v_{i-1}, v_i) < 0$
  - Let us assume, that the algorithm returns TRUE
  - In this case, we have $\forall$ i $\in$ {1, ..., k}: $d(v_i) \leq d(v_{i-1}) + c(v_{i-1}, v_i)$
  - Summing the equalities around the cycle z leads to

$$\sum_{i=1}^{k} d(v_i) \leq \sum_{i=1}^{k} d(v_{i-1}) + \sum_{i=1}^{k} c(v_{i-1}, v_i)$$

  - Since each link appears only once in the cycle z, it holds $\sum_{i=1}^{k} d(v_i) = \sum_{i=1}^{k} d(v_{i-1})$
    and we have $0 \leq \sum_{i=1}^{k} c(v_{i-1}, v_i)$
    which contradicts the above inequality $\sum_{i=1}^{k} c(v_{i-1}, v_i) < 0$
  - Thus, the algorithm returns FALSE ■

Comparison of Link State and Distance Vector Algorithms

Message complexity
- LS: with n nodes, E links, O(n·E) msgs sent each
- DV: exchange between neighbors only
  - convergence time varies

Speed of Convergence
- LS: O(n^2) algorithm requires O(n·E) msgs
  - may have oscillations
- DV: convergence time varies
  - may be routing loops
  - count-to-infinity problem

Robustness: what happens if router malfunctions?

LS:
- Node can advertise incorrect *link* cost
- Each node computes only its *own* table

DV:
- DV node can advertise incorrect *path* cost
- Each node's table used by other routers:
  - Errors propagate through network

Hierarchical Routing

Our routing study so far is an idealization:

- All routers are assumed to be identical
- Network is assumed to be "flat"

… Practice, however, looks different

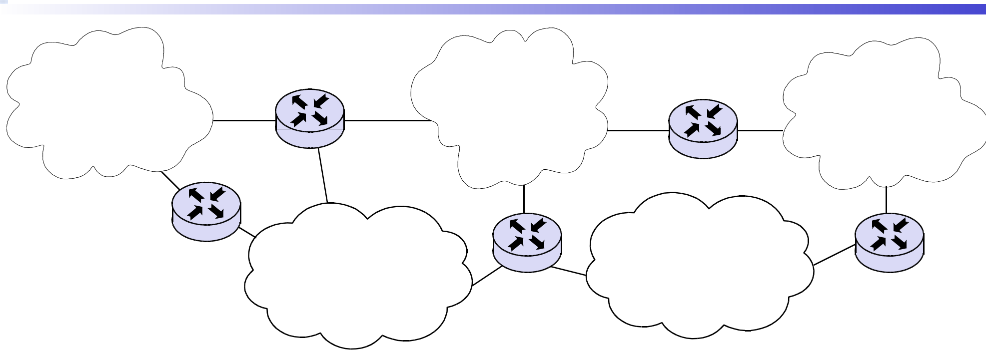
Scale (>100 million destinations!):

- Can't store all destinations in routing tables
- Routing table exchange would overload links

Administrative autonomy:

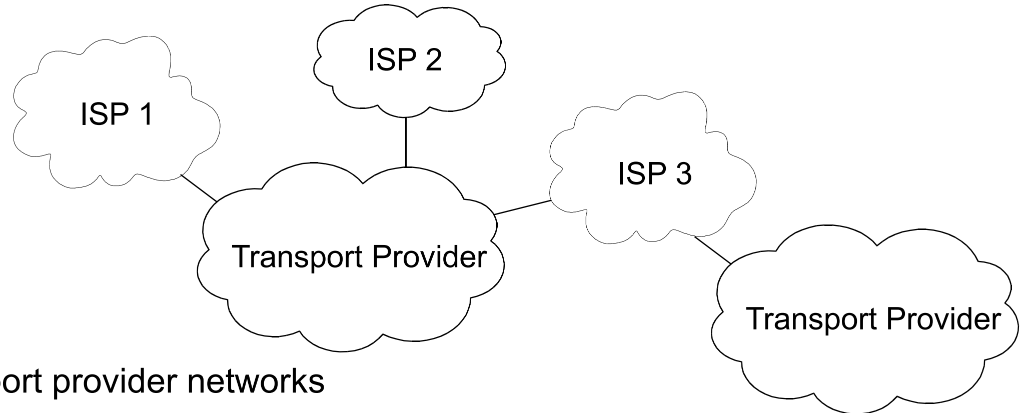
- Internet = network of networks
- Each network admin may want to control routing in its own network

Data Transmission in Interconnected Networks



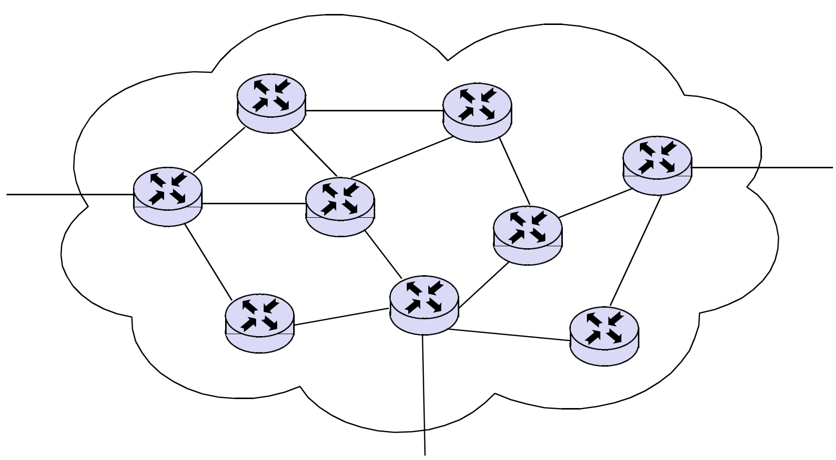
- Data transmission usually involves multiple networks
- Routing in Internet distinguishes two levels:
  - Intradomain routing inside autonomous systems (networks themselves)
  - Interdomain routing between autonomous systems (AS)
  - Internet-wide routing via border gateway protocol (BGP) operates on the AS level, every intermediate network is considered as being one hop
- Internet service providers (ISP) have peering agreements & "links"
  - If no direct link is possible, ISPs connect via a transport provider network

Transport Provider Networks



- Transport provider networks
  - Interconnect different ISP networks over longer distances
  - For close distances, ISPs also operate so-called peering points (= room full of routers from different ISPs with direct connection) in order to save costs
  - Transport providers are usually large telecommunication network operators, their transmission networks often deploy technologies as SONET, SDH, WDM, and carry multiple types of traffic (IP, voice, …)
  - ISPs can be connected to more than one transport network

Handling Traffic Inside Domains



- Every network operator (ISP, transport provider) has to make his own decisions regarding how to handle the traffic in his network:
  - Capacity of routers and links
  - Routing algorithm
  - Link costs

Notion of Traffic and Traffic Demand

- In order to handle the traffic inside his network, every network provider needs to estimate/determine the *traffic demand* for his network
- If $V = \{v_1, \ldots, v_n\}$ are the nodes (routers) in the network, then we can consider the **demand volume matrix**

  $$\mathbf{H : \{1, ..., n\} \times \{1, ..., n\} \rightarrow |N}$$

  with H[i, j] denoting the traffic demand volume between nodes v_i and v_j

  - We will also write the entry H[i, j] as h_{ij}
  - The unity of h_{ij} is not of importance for our discussion (e.g. think of Mbit/s or average sized packets per second, pps)

Considerations on Traffic Demand and Link Utilization (1)

- In order to later on understand **constraints on maximum link utilization**, we need to recapitulate some basic facts on the nature of traffic in the Internet:
  - Packets are delayed in every router of a path due to store-and-forward processing and queuing in routers
  - **Traffic congestion** can occur in parts of the Internet
  - **Packets may be dropped** if arriving at a router with **full output queues**
- Thus, the task of a network designer is to design a network in a way that:
  - **delay, congestion and probability** of packet dropping are **minimized**
  - while allowing for a **reasonable utilization** of the network
- This task becomes a bit complicated due to the fact that **traffic arrival patterns and packet sizes in the Internet are random**
  - When do people use the Internet?
  - When do applications send packets of what size?

Considerations on Traffic Demand and Link Utilization (2)

- In order to characterize Internet traffic behavior, large scale measurements are needed that can give insight on
  - traffic (inter-)arrival distribution
  - packet size distribution
- It has been observed that **Internet traffic** in fact does not follow commonly known distributions as normal distribution or exponential distribution but shows **self-similar characteristics** and can have **heavy-tailed distributions**
- For simplicity let us nevertheless assume for a moment that
  - arrive according to a Poisson process with rate λ: (on the average one arrival in every time interval of length 1/λ; which they do not in reality)

$$P_n(t) = \frac{(\lambda t)^n}{n!} e^{-\lambda t}$$

  - packet size is exponentially distributed leading to exponentially distributed service time with rate μ
  - so that the system considering one router can be thought of as the well-known M/M/1 queuing system
  - Pardon? :o)

Describing Traffic: The Poisson Process (1)

- Let A(t) (for t ≥ 0) be the number of packets arriving in the interval (0, t]. Consider the following requirements,
  1. No packet arrives at time 0: A(0) = 0
  2. **Independence** of the number of arrivals in disjoint time periods

$$Pr[A(s_1,t_1) = k_1, ..., A(s_n,t_n) = k_n] = \prod_{i=1}^{n} Pr[A(s_i,t_i) = k_i]$$

  3. **Singularity** of arrivals events (happen one after the other)

$$\lim_{s \to t} \frac{Pr[A(t) - A(s) > 1]}{t-s} = 0 \qquad (s < t)$$

  4. **Stationary process** of arrivals: the probability that n arrivals happen in a time interval (s, t) only depends on the interval length

$$\forall h \geq 0 : Pr[A(t) - A(s) = n] = Pr[A(t+h) - A(s+h) = n]$$

$$\Rightarrow Pr[A(t) - A(s) = n] = Pr[A(t-s) = n]$$

Describing Traffic: The Poisson Process (2)

- We would like to describe the arrival process A(t) mathematically
- Let $P_n(t)$ denote the probability that n packets arrive in (0, t]

$$P_n(t) = Pr[A(t) = n]$$

- As we required that no packet arrives at t = 0, that is A(0) = 0, we have

$$P_0(0) = 1 \ \text{ and } \ \forall n > 0 : P_n(0) = 0$$

- Taking advantage of the singularity of arrivals we choose Δt so small that a maximum of one arrival can happen during Δt
  - We define the rate λ(t): $\lambda(t) := \lim_{\Delta t \to 0} \frac{\sum_{i=1}^{\infty} P[A(t+\Delta t)-A(t)=i]}{\Delta t}$
  - As A(t) is stationary, we do not need to consider the point in time and have

$$\lambda = \lim_{\Delta t \to 0} \frac{\sum_{i=1}^{\infty} P[A(\Delta t)=i]}{\Delta t}$$

  - We say a function f(t) is element of the class of functions o(t) if

$$\lim_{t \to 0} \frac{f(t)}{t} = 0$$

Describing Traffic: The Poisson Process (3)

- Due to singularity of arrival events, arrival of two or more packets during Δt gets very unlikely for small Δt:

$$\sum_{i=2}^{\infty} P[A(t+\Delta t) - A(t) = i] \in o(t)$$

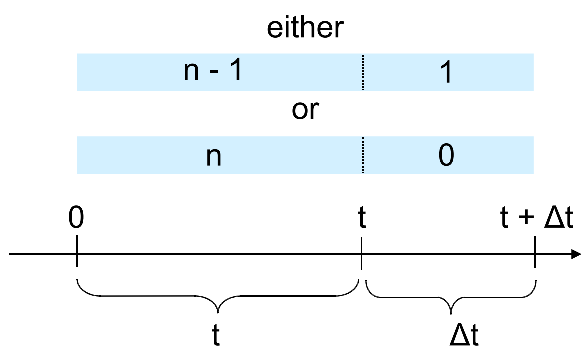
- Therefore, we get: $\lambda = \lim_{\Delta t \to 0} \frac{P[A(\Delta t)=1]}{\Delta t}$
- With this we obtain: $P[A(\Delta t) = 1] = \lambda \Delta t$

  and: $P[A(\Delta t) = 0] = 1 - \lambda \Delta t$

- Concluding, the probability of one arrival in Δt is λΔt.
- We also call **λ** the **arrival rate** of the arrival process.

Describing Traffic: The Poisson Process (4)

- We would like to use this to compute the probability of having n > 0 arrivals in the interval (0, t + Δt]
- For this, we partition the interval into two intervals (0, t] and (t, t + Δt]
- We have to distinguish two cases that both may lead to n arrivals:
  - We have n - 1 arrivals in interval (0, t] and one arrival in (t, Δt]
  - We have n arrivals in interval (0, t] and no arrival in (t, Δt]



Describing Traffic: The Poisson Process (5)

- Both cases correspond to disjoint random events and there are no more possibilities to have n arrivals in the interval (0, t + Δt]
- Thus
$$\begin{aligned} P_n(t+\Delta t) &= P[A(t+\Delta t) = n] \\ &= P[A(t) = n-1, A(t+\Delta t) - A(t) = 1] \; + \\ &\quad\; P[A(t) = n, A(t+\Delta t) - A(t) = 0] \end{aligned}$$
- Due to independence and stationarity of the arrival process, we can simplify

$$\begin{aligned} P_n(t+\Delta t) &= P[A(t) = n-1]P[A(t+\Delta t) - A(t) = 1] \; + \\ &\quad\; P[A(t) = n]P[A(t+\Delta t) - A(t) = 0] \end{aligned}$$

Describing Traffic: The Poisson Process (6)

- Thus, we get $P_n(t+\Delta t) = P_{n-1}(t)\lambda\Delta t + P_n(t)(1-\lambda\Delta t)$

  and $P_0(t+\Delta t) = P_0(t)(1-\lambda\Delta t)$
- With Δt → 0 we obtain the following two differential equations (n>0)

$$\frac{dP_n(t)}{dt} = \lambda P_{n-1}(t) - \lambda P_n(t)$$

$$\frac{dP_0(t)}{dt} = -\lambda P_0(t)$$

- We try the following substitution $u_n(t) = \frac{P_n(t)}{e^{-\lambda t}} \iff P_n(t) = e^{-\lambda t}u_n(t)$
- With this we get: $\frac{d(e^{-\lambda t}u_n(t))}{dt} = \lambda(e^{-\lambda t}u_{n-1}(t)) - \lambda(e^{-\lambda t}u_n(t))$

$$\frac{d(e^{-\lambda t}u_0(t))}{dt} = -\lambda(e^{-\lambda t}u_n(t))$$

- Using the product rule on the left side $\frac{du_n(t)}{dt} = \lambda u_{n-1}(t)$

$$\frac{du_0(t)}{dt} = 0$$

Describing Traffic: The Poisson Process (7)

- Obviously, the second differential equation requires $u_0(t) = c_0$
- So, $u_0(t) = 1$ is a constant function.
- As we know that $u_0(0) = P_0(0) = 1$ we get $u_0(t) = 1$
- For n = 1 we obtain $\frac{du_1(t)}{dt} = \lambda u_0(t) = \lambda$
- Thus, we can deduce $u_1(t) = \lambda t + c_1$
- Again, we use a constraint $u_1(0) = P_1(0) = 0$
- Which leads to $c_1 = 0$ and thus $u_1(t) = \lambda t$
- For n = 2 we obtain $\frac{du_2(t)}{dt} = \lambda u_1(t) = \lambda\lambda t$
- Again, we use a constraint $u_2(0) = P_2(0) = 0$
- Which leads to $u_2(t) = \frac{(\lambda t)^2}{2}$

Describing Traffic: The Poisson Process (8)

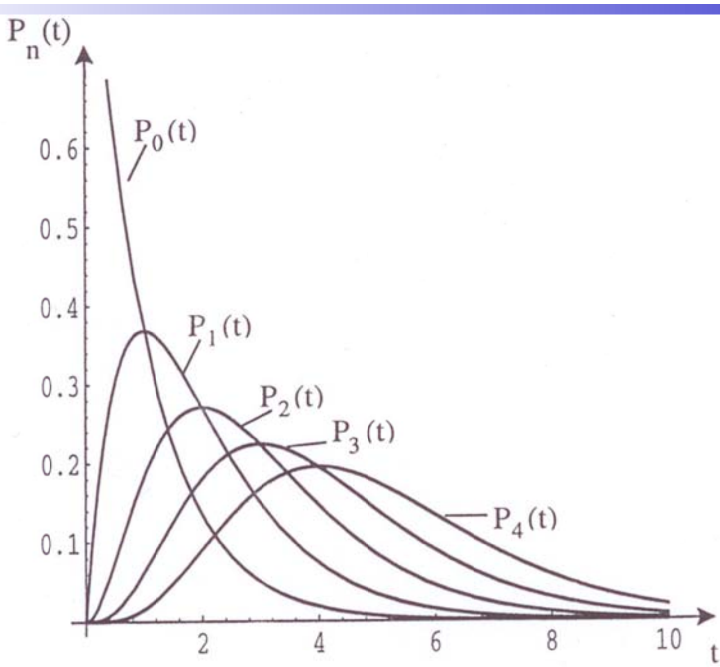
- For n ≥ 2 we obtain by induction $\frac{du_n(t)}{dt} = \lambda \frac{(\lambda t)^{n-1}}{(n-1)!}$

  and with this $u_n(t) = \frac{(\lambda t)^n}{n!}$

- Resubstituting we obtain $P_n(t) = \frac{(\lambda t)^n}{n!} e^{-\lambda t}$

- The arrival process characterized by this equation is called *Poisson Process with parameter λ*
- The set of tuples $\{(n, P_n(t)), n \geq 0\}$ is called *Poisson Distribution with parameter λt*
- The Poisson Process is of paramount importance for queuing theory

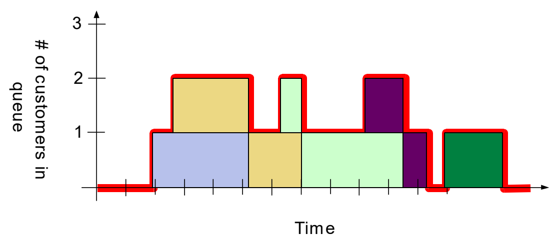
Describing Traffic: The Poisson Process (9)



The Poisson Distribution $\{P_n(t), n \geq 0\}$ with $\lambda = 1/min$

Little's Law (1)

- Let Arrival(T) be the number of packets arrived until time T,
  Wi(T) be the waiting time of packet i at time T
  N(T) be the number of packets in the system at time T
- We are interested in the accumulated (total) waiting time of all jobs that ever arrived in the system until time T
- This can be computed via two observations:
  - Sum of the waiting times of all packets arrived until time T: $$\sum_{i=1}^{Arrival(T)} W_i(T)$$
  - Integral over the number of packets in the system N(T) during the interval (0, T] $$\int_{t=0}^{T} N(t)dt$$



Little's Law (2)

- Obviously both computations have to lead to the same result

$$\sum_{i=1}^{Arrival(T)} W_i(T) = \int_{t=0}^{T} N(t)dt$$

- For Arrival(T) > 0, we can extend the equation as follows

$$\frac{Arrival\,(T)}{T} \cdot \frac{1}{Arrival\,(T)} \sum_{i=1}^{Arrival(T)} W_i(T) = \frac{1}{T} \int_{t=0}^{T} N(t)dt$$

- Let λ(T) be the average number of packets in time (0, T]

$$\lambda(T) = \frac{Arrival\,(T)}{T}$$

Little's Law (3)

- Let $\overline{W}(T)$ be the average waiting time of a packet

$$\overline{W}(T) = \frac{1}{Arrival\,(T)} \sum_{i=1}^{Arrival(T)} W_i(T)$$

- Let $\overline{N}(T)$ be the average number of packets in the system

$$\overline{N}(T) = \frac{1}{T} \int_{t=0}^{T} N(t)dt$$

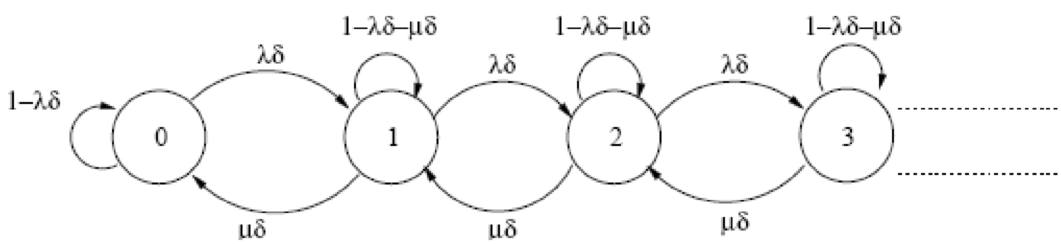
- Then we can write the above equation as: $\lambda(T) \cdot \overline{W}(T) = \overline{N}(T)$
- If the system is in stationary condition, that is the number of packets entering is equal to the number of packets leaving the system, then the following limits do exist

$$\lim_{T\to\infty} \lambda(T) = \lambda \qquad \lim_{T\to\infty} \overline{W}(T) = \overline{W} \qquad \lim_{T\to\infty} \overline{N}(T) = \overline{N}$$

- With this we get **Little's Law**: $\lambda \cdot \overline{W} = \overline{N}$

Solving the M/M/1 System (1)

- Let us consider the number of packets in a router if packet arrivals are Poisson distributed with rate λ and service times exponentially distributed with rate μ
- In the diagram below the state identifiers denote the number of packets in the system and δ denotes discrete times



- The diagram describes the state changes of the system, and it is generally referred to as a Markov chain (as state changes are only dependent on finitely many prior states)

Solving the M/M/1 System (2)

- Let p_n denote the probability of the system being in state n
- Then in case of statistical balance between states, we can formulate the following equation for all states n: $p_n \mu \delta = p_{n-1} \lambda \delta$

  $$\Rightarrow p_n = \frac{\lambda}{\mu} p_{n-1} = \left(\frac{\lambda}{\mu}\right)^n p_0 = \rho^n p_0$$

  with $\rho = \frac{\lambda}{\mu}$ denoting the utilization of the system

- Recall: $\lim_{n \to \infty} \sum_{i=0}^{n} p_i = \lim_{n \to \infty} \sum_{i=0}^{n} \rho^i p_0 = p_0 \frac{1}{1-\rho}$ (geometric sum)
- As furthermore, $\lim_{n \to \infty} \sum_{i=0}^{n} p_i = 1 \Rightarrow p_0 = 1 - \rho$

  we obtain $p_n = (1 - \rho)\rho^n$

  (this also holds if we consider $\lim_{\delta \to 0}$)

Solving the M/M/1 System (3)

- Let us now consider the average number N of packets in the system

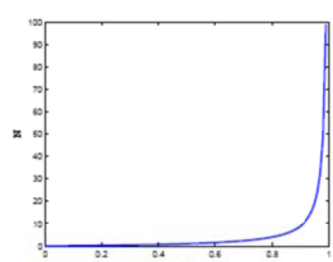
  $$\overline{N} = \sum_{i=0}^{\infty} i p_i = \sum_{i=0}^{\infty} i \rho^i (1-\rho) = \frac{\rho}{1-\rho} = \frac{\frac{\lambda}{\mu}}{1-\frac{\lambda}{\mu}} = \frac{\lambda}{\mu - \lambda}$$

- As Little's Law states $\lambda \overline{W} = \overline{N}$

  we obtain for the average waiting time $\overline{W} = \frac{1}{\mu - \lambda}$

- Note that with $\rho \to 1$ we will have

  $\overline{W} \to \infty$ and $\overline{N} \to \infty$



- Let us resume our considerations on traffic demand and link utilization

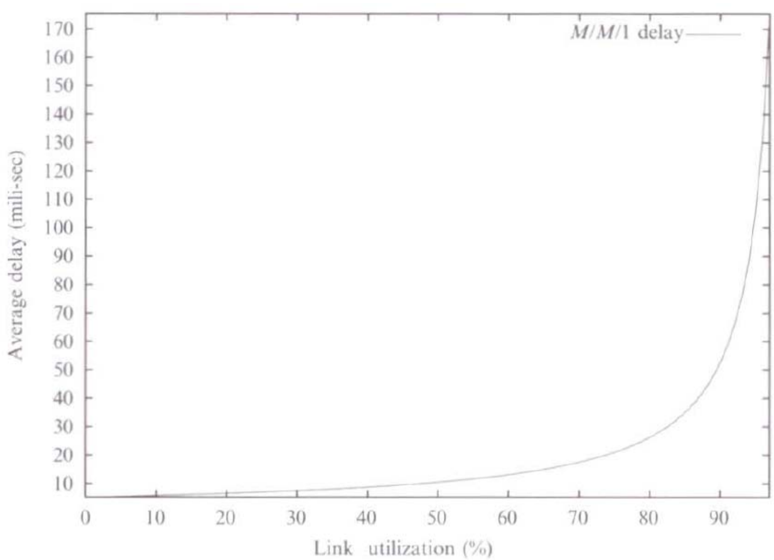
Considerations on Traffic Demand and Link Utilization (3)

- If packets have average size K_p bits and link capacity is C bits per second then the average service rate of the link is $\mu_p = C / K_p$ pps (packets per second)
- If the average arrival rate is λ_p pps then the average delay is given by

$$D(\lambda_p, \mu_p) = \frac{1}{\mu_p - \lambda_p}$$

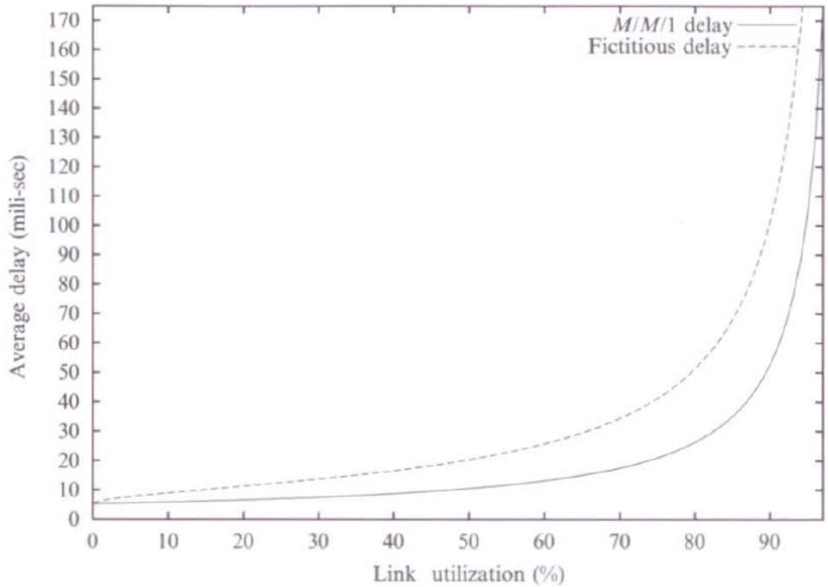
- Even if vastly simplified (due to our simple traffic assumptions), this can provide useful insights on delay
  - Consider a T1-link with 1.54 Mbit/s, then for an average packet size of 1 kByte = 8 kbit the average service rate of the link is 190 pps
  - If the packets arrive with rate λ_p=100 pps, then the average delay is 1 / 90 ~ 11.11 ms
  - If the arrival rate is increased to 150 pps, the delay increases to 25 ms
- Let us also consider the average link utilization $\rho = \lambda_p / \mu_p$
  - For λp = 100 pps and μp = 190 pps, we have ρ = 0.526

Considerations on Traffic Demand and Link Utilization (4)



- The above curve shows the average delay of an M/M/1 queuing system (for a given arrival rate λ) as a function of link utilization ρ
- So, in order to keep the average delay below 15 ms the link utilization should be kept below 64.5%

Considerations on Traffic Demand and Link Utilization (5)



- As in reality Internet traffic does not arrive according to a Poisson process, link utilization should be kept even lower, e.g. below 50%

TELEMATIK Rechnernetze

Considerations on Traffic Demand and Link Utilization (6)

- So, when link utilization reaches a certain threshold, e.g. 50%, it should be upgraded
- However, from a delay perspective, it is better to have one high bandwidth link than multiple lower bandwidth links
  - Consider a tenfold increase in both arrival rate and service rate:

$$D(10\lambda_p, 10\mu_p) = \frac{1}{10\mu_p - 10\lambda_p} = \frac{1}{10} \cdot \frac{1}{\mu_p - \lambda_p}$$

  - So the average delay is reduced to one tenth
  - This is often referred to as the **statistical multiplexing gain**
  - Given the typical cost structure of low vs. high bandwidth links, it is even more beneficial to have one high bandwidth link
- On the other hand, delay is not the only characteristic to consider
  - Fault-tolerance requirements may call for having multiple links
  - On a single link misbehaving traffic flows are difficult to control

Considerations on Traffic Demand and Link Utilization (7)

- So, if we are able to predict or measure the utilization of a single link, then we can decide to upgrade the link once its utilization reaches a certain threshold
- However, in a network consisting of multiple routers and links, this gets more complicated:
  - Link utilization is also influenced by routing decisions and the utilization of other router's and links
  - Routing decisions might be influenced by
    - delay experienced by packets,
    - average queue length in routers (over a recent period of time),
    - currently available link capacities etc.
- What if capacities of links are not pre-determined?
  - Can link capacity dimensioning and routing decisions be optimized in a joined way?
  - How to account for fault-tolerance requirements when doing so?

Notion of Routing and Flows

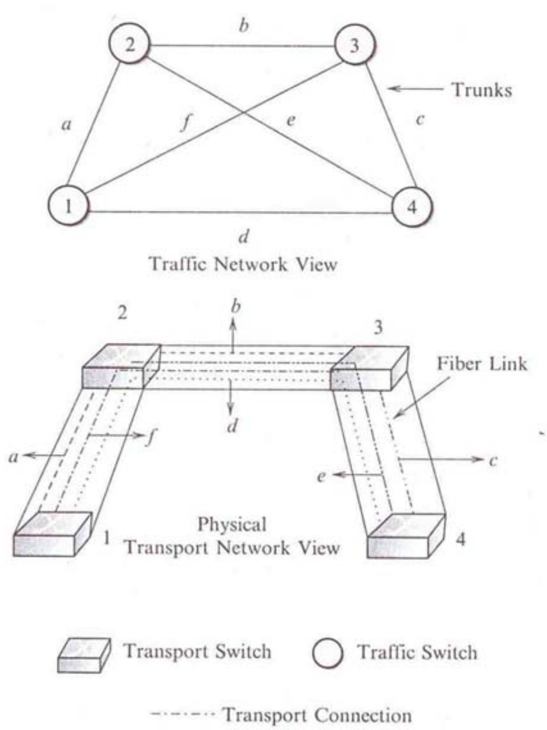
- Up to now, we have used the word routing in the context of making routing decisions for individual packets
- However, there are **two different ways to interpret** the term **route**:
  - How an individual packet may be transported in the networks
  - How, in general, ensemble traffic may be routed between the same two points (e.g. all packets flowing from New York to Berlin)
- From now on and for the remainder of this course, we will stick to the second notion of route and taking routing decisions unless we explicitly state that we mean the first notion
- So, we are more interested in **making routing decisions for flows** of packets, for which we have a (more or less accurate) traffic description (e.g. constant bit rate, Poisson arrival with rate λ etc.)
- These routing decisions will
  - have to stay within capacity constraints,
  - in some cases influence capacity decisions (joined routing/dimensioning)

Multi-Level Networks (1)

- As we have seen before, sometimes ISPs need to interconnect their networks via a transport network
  - The transport network, in general, may use a different protocol architecture, e.g. SONET, SDH, ATM
  - From the point of view of the transport network provider, his client (an ISP) demands a certain transport volume, e.g. expressed
    - Simply in MBit/s between two or more points, or more fine grained by
    - Leaky bucket with rate r, burst size b, min. & max. packet size
- The resulting overall network architecture is a **multi-level architecture**, consisting of two views
  - **Transport view:** network with protocol architecture X (SONET, SDH, …)
  - **Traffic view:** network with protocol architecture Y (IP, ISDN, …)

Multi-Level Networks (2)

- Links in the traffic network are **logical links**
  - Logical links have to be mapped to links/paths in the transport network
  - This mapping can change the properties of the network from one view to the other
- Consider **path diversity**:
  - In the traffic view there are three link-diverse paths from node 1 to node 4
  - In the transport view all three logical links make use of the same transport link
- This has **implications on protection and restoration design**



Chapter Summary

- Data traffic is usually transported in packets that are individually forwarded through interconnected routers in the network
- Routing decisions can be guided by minimizing the total “cost” of a path summing up individual link costs, and we know well established routing algorithms for this: Dijkstra’s Algorithm, Bellman-Ford Algorithm, Distance Vector Routing
- Traffic can be characterized according to a stochastic process:
  - The Poisson Process is a well established model and it shows ideal characteristics: independence, singularity, stationarity
  - Real Internet traffic looks different though (self similar characteristics)
- Average link load should not exceed a certain threshold (e.g. 50%), otherwise long average delay occurs
- Routing decisions heavily influence link utilization and should take traffic demand, link capacities, etc into account
- In multi-level networks, characteristics may change between views