Interconnection topologies (cont.)

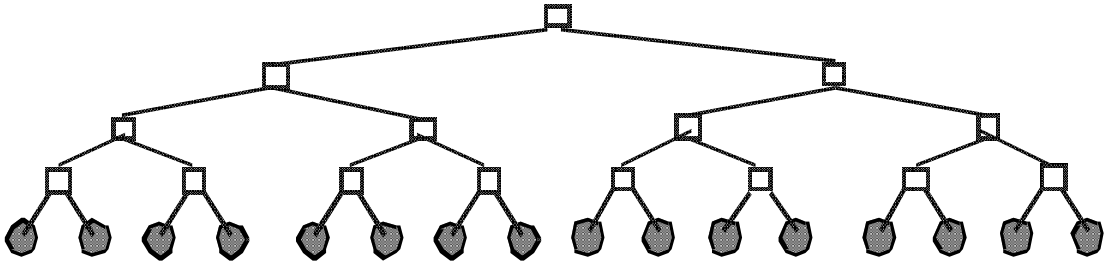
[§10.4.4] In meshes and hypercubes, the average distance increases with the dth root of N.

In a tree, the average distance grows only logarithmically.

A simple tree structure, however, suffers from two problems.

- Congestion

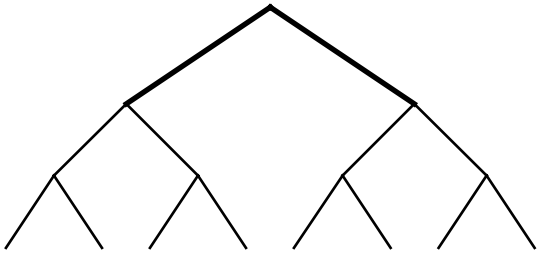
- Its fault tolerance is low.



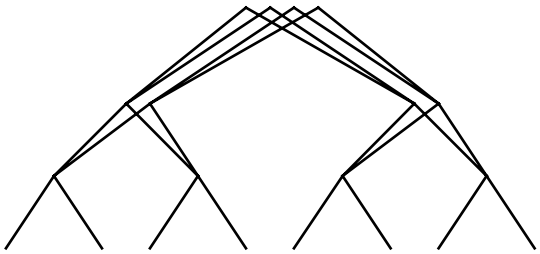
9. Fat trees

One approach to overcoming the limitations of the tree topology was devised by Leiserson and implemented in the Thinking Machines CM-5 data network.

The idea is that the edges at level k should have two or more times the capacity of the edges at level k+1 (the root is at level 0).



In reality, the links at higher levels are formed by replicating connections.



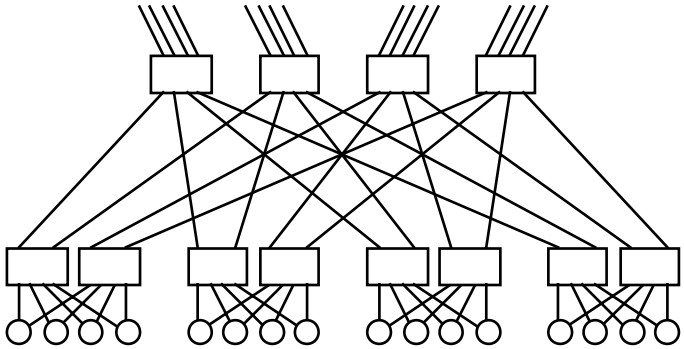
The algorithm for routing a message from processor *i* to processor *j* is as follows:

- Starting from processor *i*, a message moves up the tree along the path taking it to the first common ancestor of *i* and *j*.
- There are many possible paths, so at each level the routing processor chooses a path at random, in order to balance the load.
- Upon reaching the first common ancestor, the message is then routed down along the unique path connecting it to processor *j*.

The diameter of a fat tree is

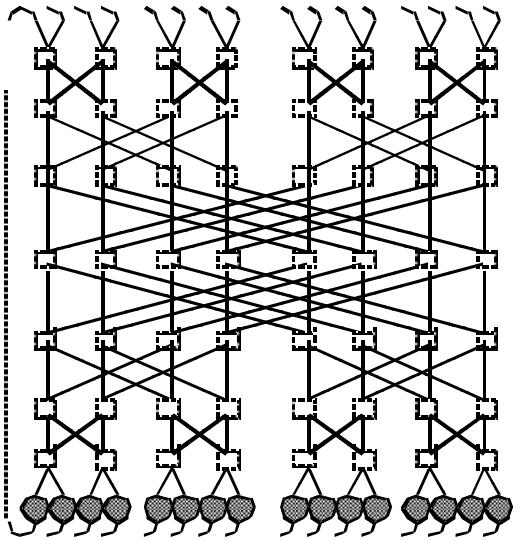
and its bisection width is

We have shown a fat tree based on a binary tree. It may also be based on a *k*-ary tree. The CM-5 uses fat trees based on 4-ary trees:

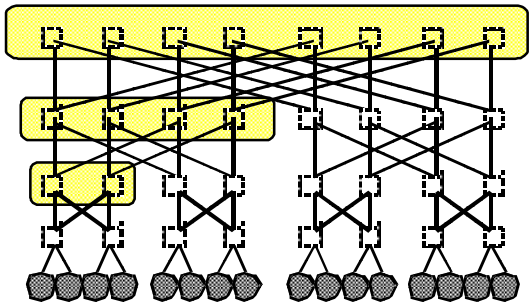


A *k*-ary fat tree can also be viewed as a *k*-ary Benes network that is folded back on itself in the high-order dimension:

16-node Benes Network (Unidirectional)



16-node 2-ary Fat-Tree (Bidirectional)



The collection of $N/2$ switches at level i is viewed as 2^{d-i} "fat nodes" of 2^{i-1} switches, where d is the *dimension* of the switch (where d is the number of levels in the tree—4 in the picture).

Routing algorithms

[§10.6] What path does a message travel from its source to its destination? This is determined by the routing algorithm.

Given a current node and a destination node, the routing algorithm chooses the next *port* and *channel* on which to send out the message.

Thus, a routing algorithm is a function $R: N \times N \rightarrow C$.

A switch usually uses one of three mechanisms to determine the output channel from info in the packet header.

- arithmetic,
- source-based port select, and
- a table lookup.

A switch needs to route a packet every few cycles, so it needs to be fast.

In regular topologies, simple arithmetic suffices.

Example: Δx, Δy routing in a grid.

West ($-x$)	$\Delta x < 0$
East ($+x$)	
South ($-y$)	$\Delta x = 0$, $\Delta y < 0$
North ($+y$)	
Processor	$\Delta x = 0$, $\Delta y = 0$

To accomplish this routing, the switch needs to test the address in the header and increment or decrement one routing field.

Usually, routing is done in *dimension order*—first across the *x*-dimension, then the *y*-dimension, then the *z*-dimension (if any), etc.

So in a binary cube, the switch figures out which is the most significant position where the destination node number differs from the current node number.

Sometimes a packet header has a *relative address* embedded in it.

Example: If the source node is 001010 and the destination node is 100101, what would be the relative address embedded at the source node?

In this case, the switch just looks for the first non-zero bit and routes accordingly.

In general, in a mesh or cube, routing is done by moving from lowest (*x*) dimension to the highest.

This is called routing in *dimension order*. In a hypercube, it is called *e-cube* routing. It is used in nCube hypercubes and the Intel Paragon, among others.

More generally, the source builds a header consisting of the output port # for each switch along the route.

This is called *source-based* routing.

Each switch just strips off the port number from the front of the message and sends it on.

All of the intelligence is at the source node. Arbitrary routing can be supported without much logic in the switches.

Disadvantage: Header is large, of variable size.

Used in Meiko CS-2 and Myrinet.

Table-driven routing associates a small routing table at each switch. It allows for a small fixed-size header.

In table-driven routing,

- The packet header contains a routing field i.
- The output port is $R[i]$,where R is the routing table.
- Usually the table also contains the routing field for the next step in the route.

Disadvantage: Switch must contain quite a bit of routing state. Fairly large tables are needed even for simple routing algorithms.

This approach is used by ATM and HPPI switches, but isn't too practical for multiprocessors, because of the large number of routing patterns that they must support.

One important difference between network routers and multiprocessor switches: Time constraints.

Routing may be—

- *Deterministic*, where the route is completely determined by the (source, destination) pair, and not by the intermediate state, or
- *Adaptive*, where the route is influenced by traffic along the way.

A routing algorithm may be *minimal*, meaning that it only selects shortest paths toward the destination (no backtracking), or *nonminimal* (can allow longer paths).

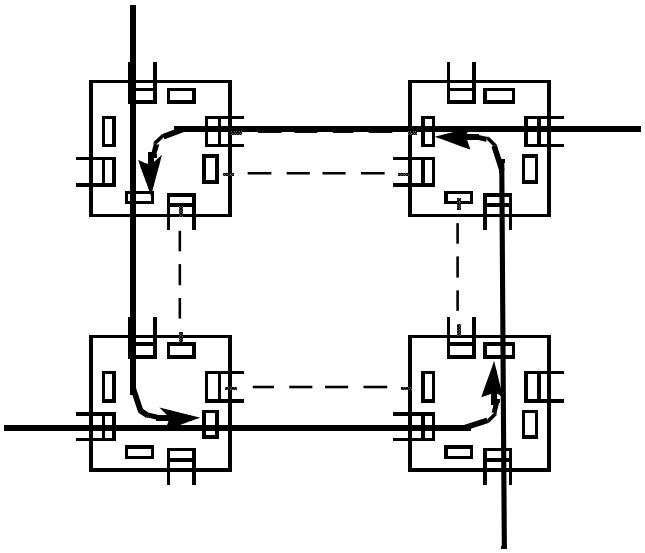
Do nonminimal algorithms have any advantages over minimal algorithms?

Do minimal algorithms have any advantages over nonminimal algorithms?

Deadlock-free routing

Deadlock occurs when two or more packets are "circularly" waiting for resources that are held by other packets in the group.

The diagram at the right illustrates how this can occur. Each packet is waiting for a link occupied by another packet.



What conditions are necessary for deadlock to occur?

- a *shared resource*
- that is *incrementally allocated* and
- non-preemptible.

A channel is a shared resource, and channels are acquired incrementally, as a route is built up.

How can deadlock be avoided? Basically, by constraining how channel resources are allocated. Routing in dimension order is one way. How can we see that in the diagram above?

How do we prove that an algorithm is deadlock free?

- Resources are logically associated with channels.
- Messages introduce dependences between resources as they move forward.
- We need to articulate possible dependences between channels,
- then show that there are no cycles in channel dependence graph

We can do this by finding a numbering of channel resources such that every legal route follows a monotonic sequence

$\Rightarrow$ no traffic pattern can lead to deadlock

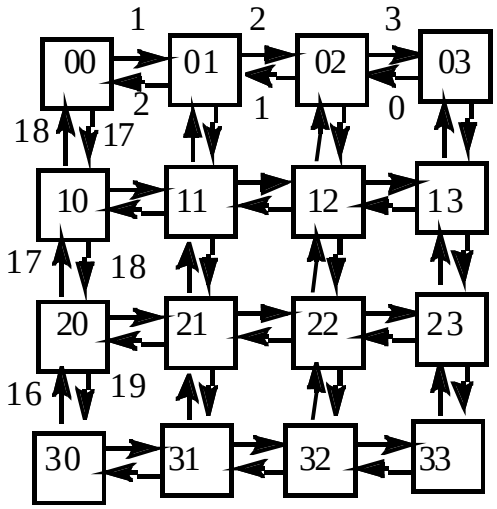
This is trivial for acyclic networks, such as shuffle-exchange and butterfly networks.

It is also trivial for trees and fat trees, as long as the upward and downward channels are independent.

Example: Show $\bullet x$, $\bullet y$ routing on a *k*-ary 2D array is deadlock free.

We view each bidirectional channel as a pair of unidirectional channels numbered independently.

Here's how we do the numbering.



- Every $+x$ channel $(i, y) \rightarrow (i+1, y)$ is numbered i.
- Similarly for $-x$ starting from 0 at the most positive edge.
- Every $+y$ channel $(x, j) \rightarrow (x, j+1)$ is numbered $N+j$.
- Similarly for $-y$ channels.

Given this numbering, any routing sequence that starts out in the *x* direction, turns and then goes in the *y* direction is increasing.

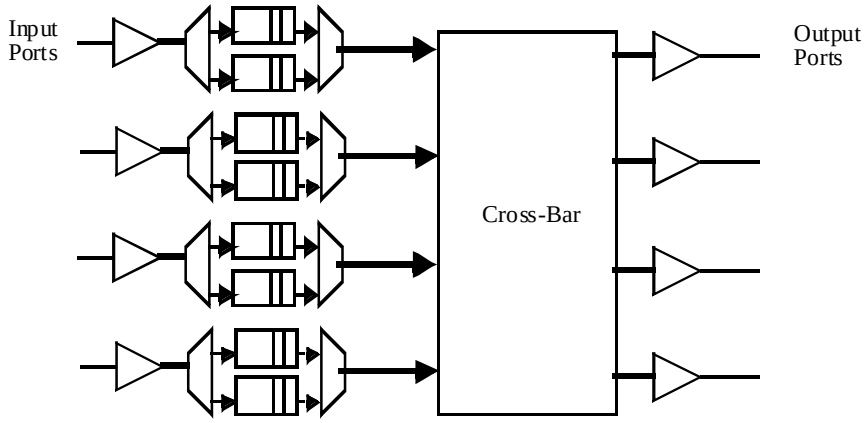
Virtual channels

This proof easily generalizes to hypercubes—but not to toruses because of the wraparound edges.

So, how can we do deadlock-free routing on toruses and other higher-order networks?

The basic approach is to provide *virtual channels*.

To do this we need more than one packet buffer per channel.



Example: Consider a torus with unidirectional links.

Reserve one packet buffer on each channel for messages destined for nodes with a higher number than their source, i.e., messages that don't use wraparound channels.

Now such messages will always be able to make progress.

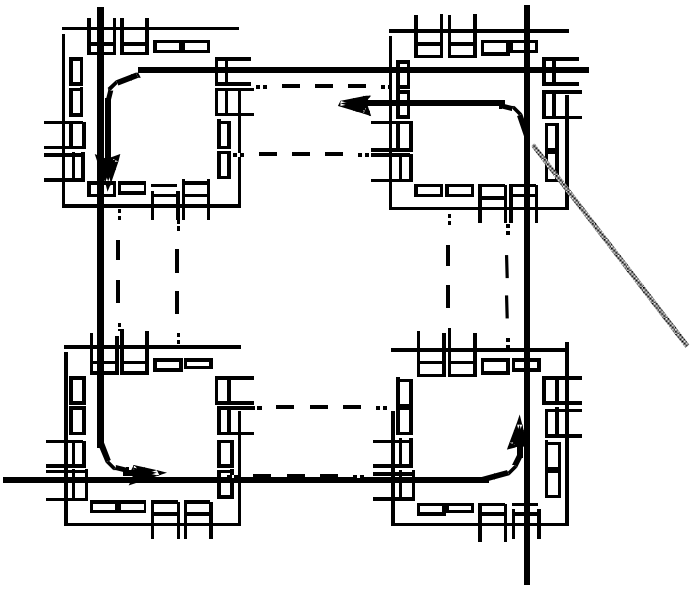
Wraparound messages may be delayed, but the network will not deadlock.

Notice that this doesn't require more links or switches, just more buffers.

How can we break the four-message deadlock shown above?

Let’s provide two virtual channels per physical channel

- Messages at a node numbered higher than their destination are routed on the “high” channels.
- Messages at a node numbered lower than their destination are routed on the “low” channels.

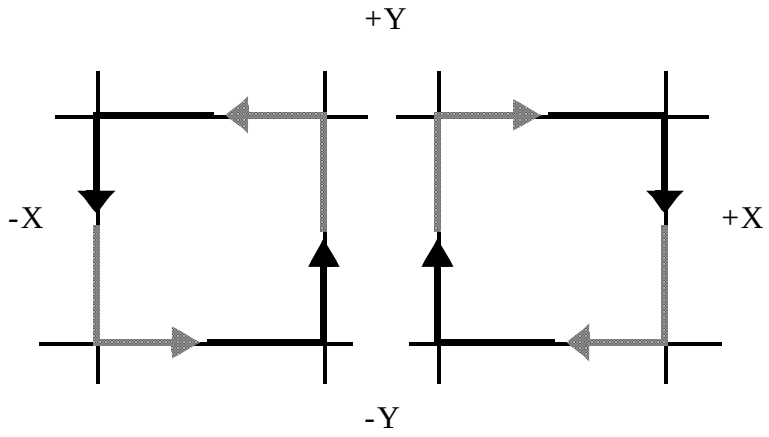


This breaks the dependence cycle.

This strategy can easily be generalized to *d* dimensions; see §10.6.5.

Turn-model routing

Note that *x-y* routing in dimension order allows only four of the eight possible turns a packet might make on its way to its destination.

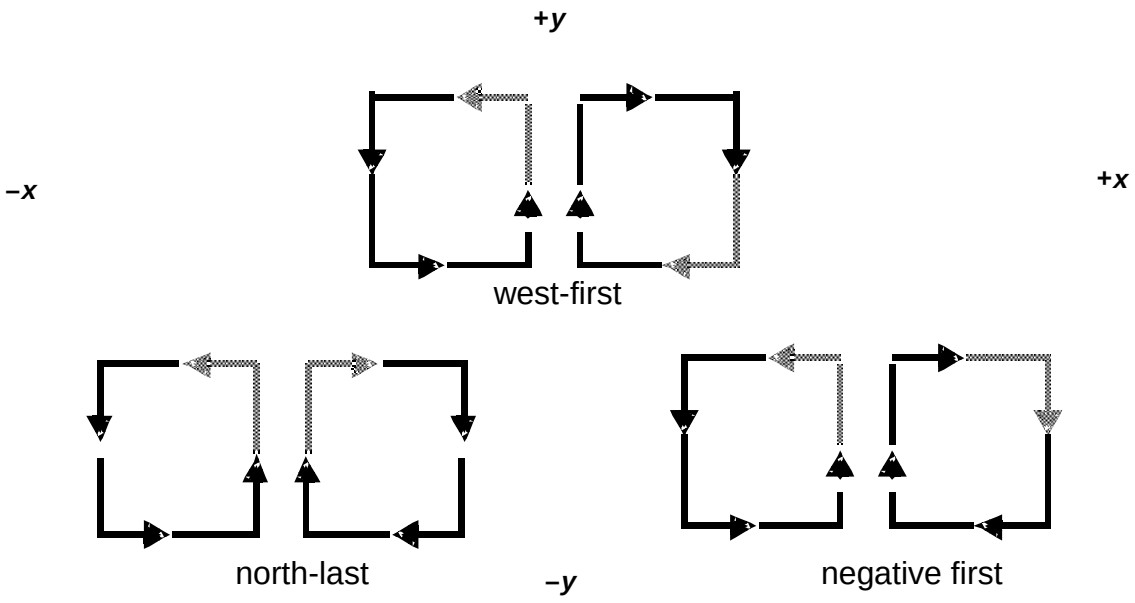


When a packet is traveling in the $\pm x$ direction, it is legal to turn into the $\pm y$ direction, but once it is traveling in the $\pm y$ direction, it can make no further turns.

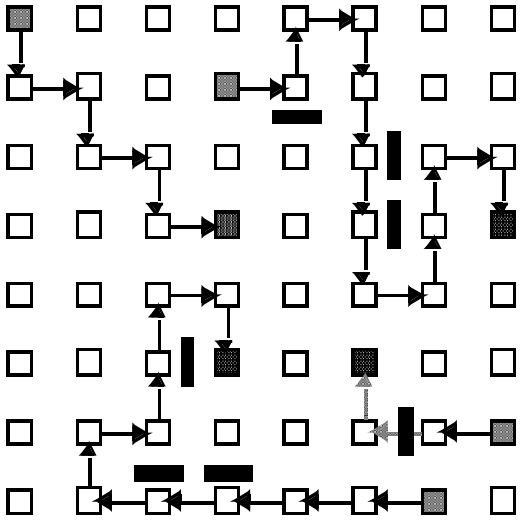
Intuitively, we could prevent deadlock by disallowing only one turn in each cycle.

It turns out that of the 16 possible ways to prohibit two turns in a 2D array, 12 prevent deadlock.

These consist of the three algorithms below, and rotations of them.



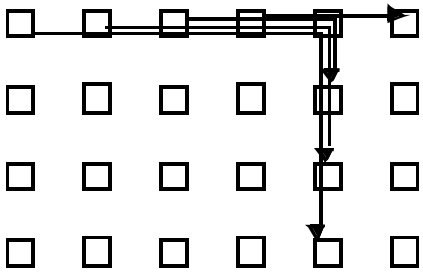
Each of these algorithms allows nonminimal routes to be followed. Here are some legal west-first routes.

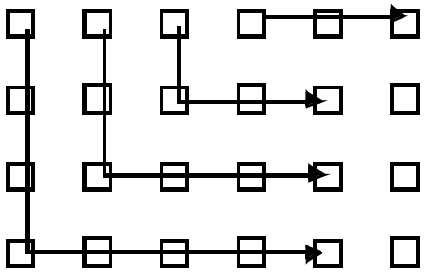


Adaptive routing

Adaptive routing has several advantages.

- If there is only one route from source to destination, failure of a single link can leave the network disconnected.
- It can allow messages to avoid regions where there are long queues and a lot of contention.





One interesting adaptive algorithm is “hot-potato” routing.

A switch never buffers packets. If > 1 packet is destined for the same output channel, the switch “misroutes” all but one.