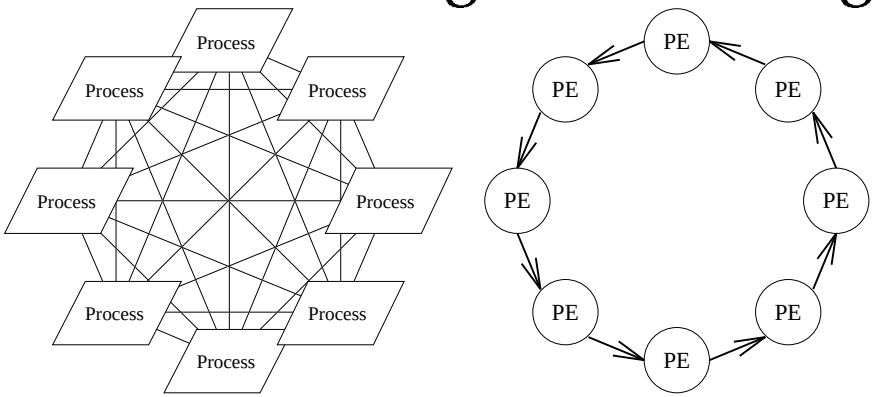
Software Message Forwarding



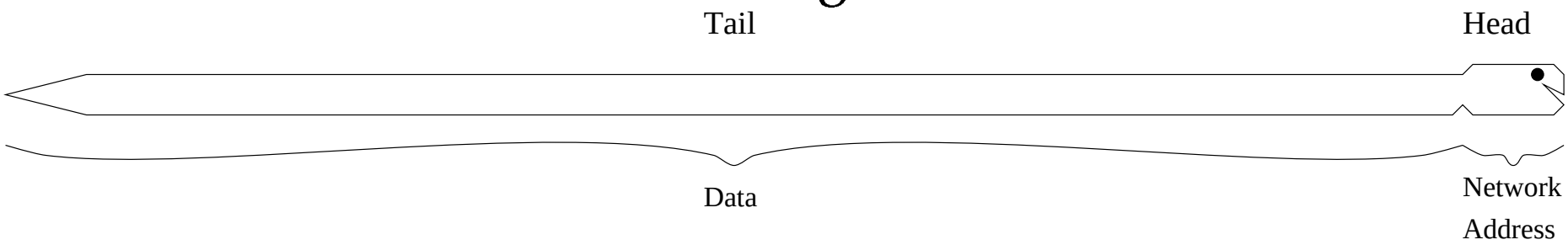
Consider the mapping of *any to any* process connectivity onto the simplest partial network.

- We must prepend a processor address to our message.
- Messages arriving at a node, carrying the local node address are consumed
- All other messages are forwarded to the next node.

This message delivery system is *Store and Forward*.

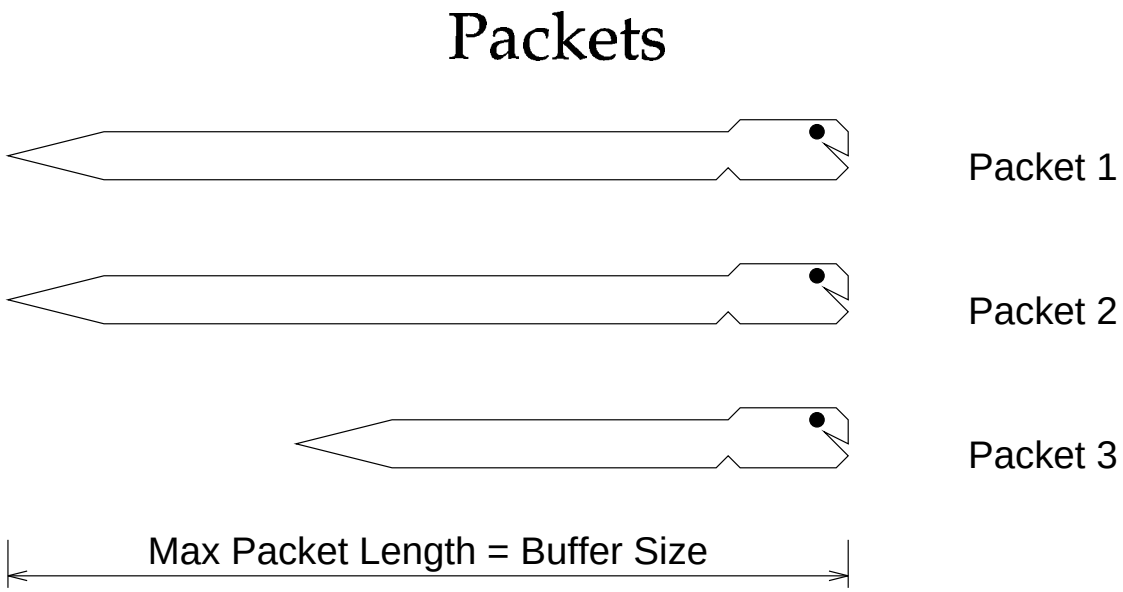
MIMD Message Forwarding

Messages



- The message consists of a header containing a unique network address and a tail consisting of the message data.
- The header may include other information such as:
  - Length of message
  - Type of message
  - Return address
- We must provide a buffer at each node which is big enough to hold the largest message.

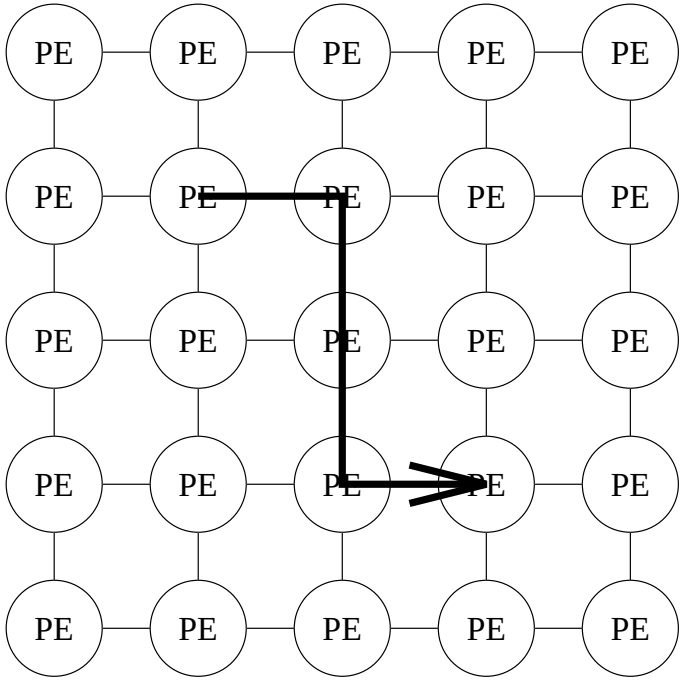
MIMD Message Forwarding



Divide message before transmission into packets.

- Each packet carries its own copy of the destination address.
- Packets have a fixed maximum length.
- On arrival the packets are re-assembled into the original message.

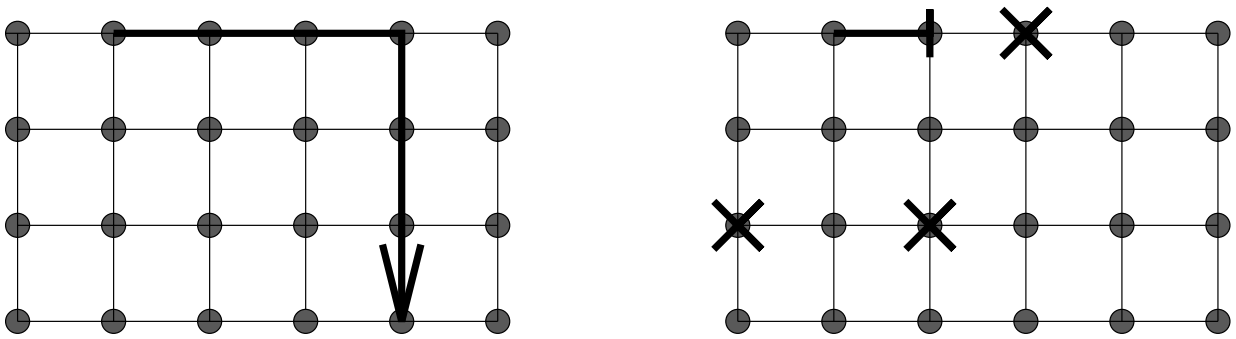
MIMD Message/Packet Routing



For more complex partial networks we must make routing decisions:

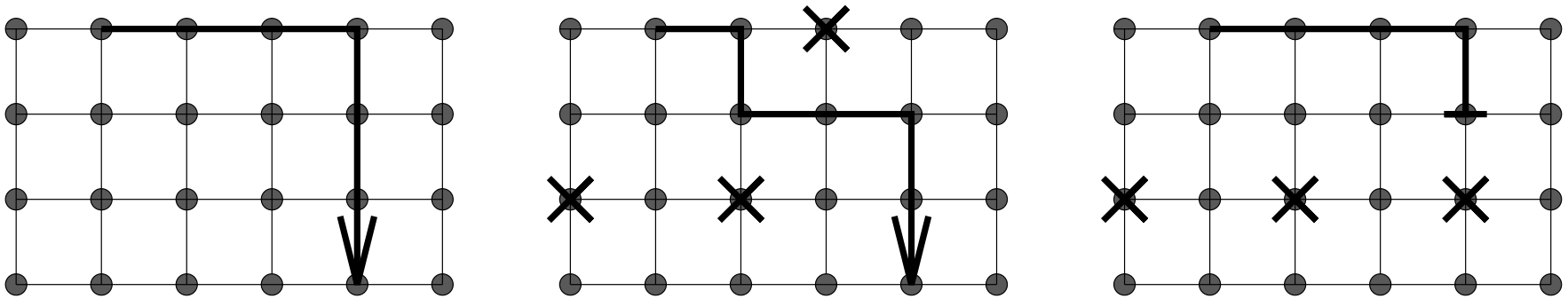
- At each node we must decide which output link should be used for each unconsumed message.

MIMD Message/Packet Routing



- Deterministic Routing
  - The path of the message is deterministic, predefined by the source and destination addresses.
  - *e.g. Route first in X and then in Y.*
  - If there is a temporary blockage at a node en route, the message must wait.

MIMD Message/Packet Routing



- Adaptive Routing
  - The path is not predetermined, at any node the message may take any link which brings it closer to it's destination.
  - The messages are sometimes able to adapt their paths in order to avoid blockages.

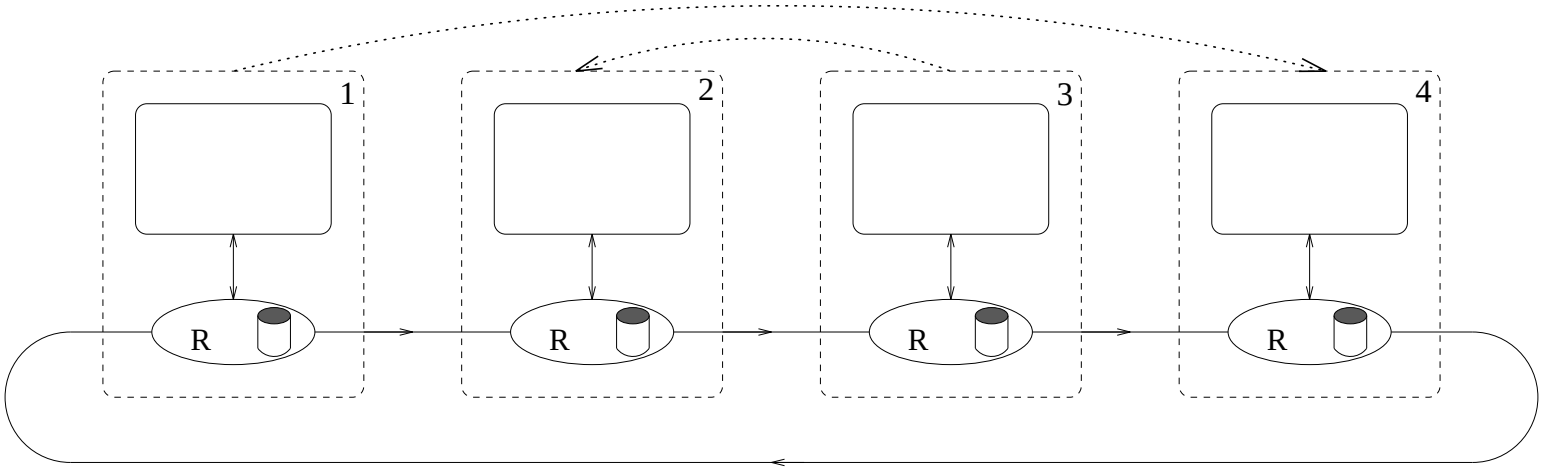
Adaptive Packet Routing

Adaptive routing may result in problems for message re-assembly. It is possible for packets to arrive out of sequence, having taken different paths to their destination.

- Packet headers may include an index number such that they can be re-ordered on arrival.
- Alternatively we can delay sending a packet until the previous packet has arrived.

  This requires the sending of an acknowledge message for each packet.

Blockages & Buffering



- Process 1 wishes to send data to Process 4, Process 4 is ready to accept data.
- Process 3 wishes to send data to Process 2, Process 2 is ready to accept data.
- Process 1 injects two segments which occupy the routing buffers 1 & 2.
- Process 3 injects two segments which occupy the routing buffers 3 & 4.
- All buffers are full, no packet has arrived, we have *deadlock*.

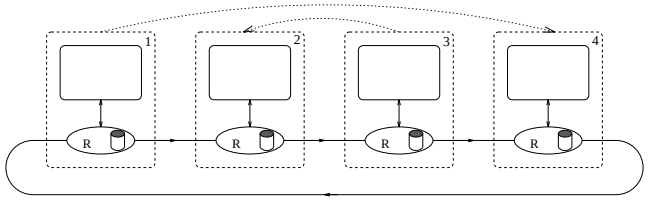
Deadlock Free Message/Packet Routing

We must distinguish between *program deadlock* and *routing deadlock*.

- We have *program deadlock* if we have packets in the network which are not being removed because the receiver process is not ready. The network may then help to spread the blockage such that all processes are stopped.
- We have *routing deadlock* where there are no arrived packets and we have cycles of blocked packets.

Our programming model assumes that we can freely send data from any process to any other process. We must have a deadlock free routing system which remains deadlock free regardless of load.

Deadlock Free Message/Packet Routing



- Let us increase the buffering.
  - The more that buffering is increased the less likely the system is to deadlock.
  - In any real system there must be an upper bound on the buffering.
  - It will always be possible for deadlock to occur.
- Let us limit the number of packets in the network[1].
  - If there are fewer packets than buffers it will always be possible for at least one packet to move.
  - We can accomplish this by having a fixed number of empty packets in the network after initialization.

[1] This is the technique used by the Cambridge Ring.

– Empty packets can be swapped for full packets and full packets for empty packets but no packets can be added or taken away.

Deadlock Freedom in other networks

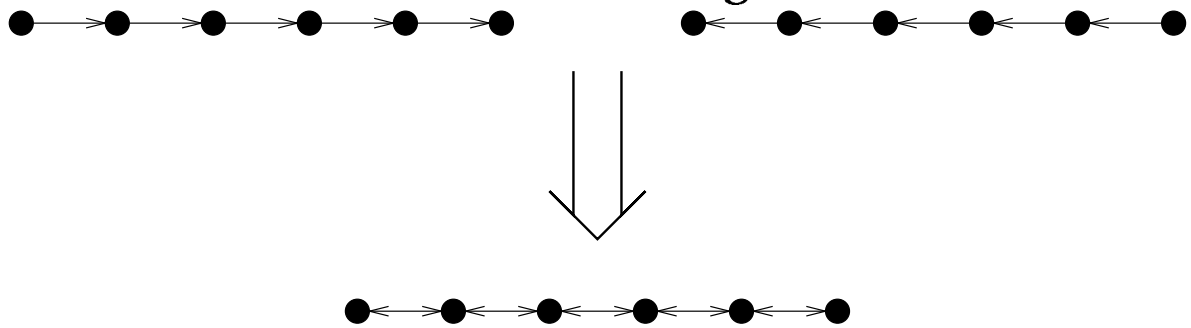
If we try to avoid deadlock by limiting packet numbers in any network with bidirectional links we find that the maximum number of packets we can use is one. We must find a new method.

Cycle Free Networks

- These uni-directional line networks are cycle free and hence deadlock free.
- Neither network is capable of providing our full connectivity.
- If we provide both networks for the connection of a single line of processors we can provide full connectivity.
- Each message will either travel in one network *or* the other.

Virtual Networks

Instead of providing two different networks we can provide two *independent virtual networks* within a single network.



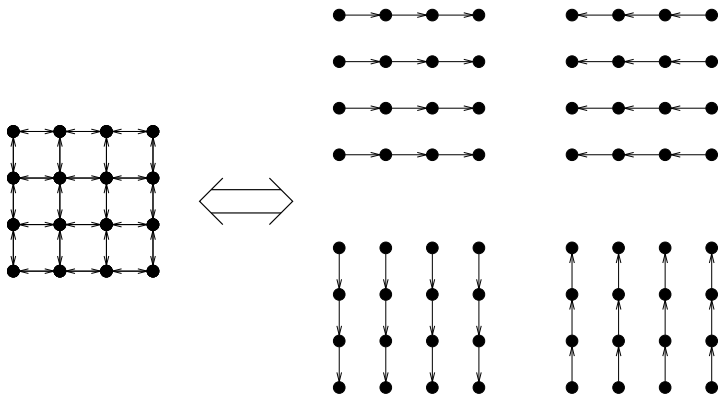
In order to maintain the independence of the virtual networks:

- Each network must have a separate set of buffers.
- Where two networks use the same link[2] they must be multiplexed such that a blockage in one network does not prevent the other network from using the link.

[2] this is not the case here

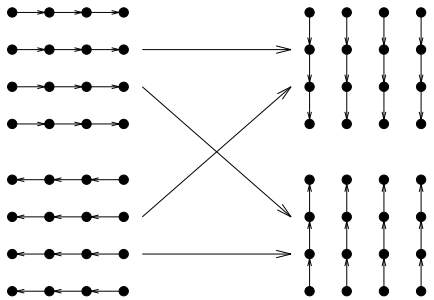
Deadlock Freedom in 2 Dimensions

We can divide a 2D bi-directional grid network into four cycle free networks.



- These four networks use different links and can be provided with separate buffer spaces.
- While they remain independent they cannot provide full connectivity.
- We must route a message in more than one network.

Deadlock Freedom in 2 Dimensions



- If we allow a message to travel from one virtual network to another network without restriction, we can re-introduce cycles and deadlock.
- If we apply an ordering to the networks and restrict the messages to travelling through the networks in order we can still avoid cycles.

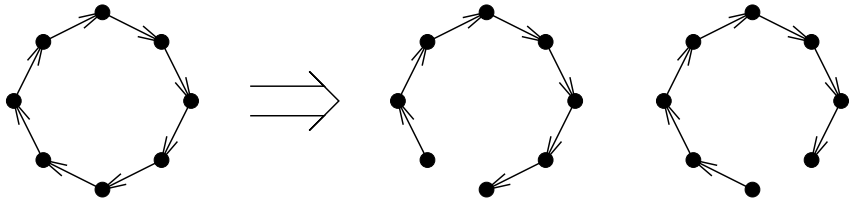
Thus we employ restricted routing:

Route first in X and then in Y.

In this case we cannot have adaptive routing.

Deadlock Freedom in Closed Networks

- So far we have applied virtual network routing to open networks.
- How shall we eliminate cycles in closed networks?



- We can map two or more open networks onto a single closed network to provide the required connectivity.
- Here we will usually need to run two links where there was previously only one.

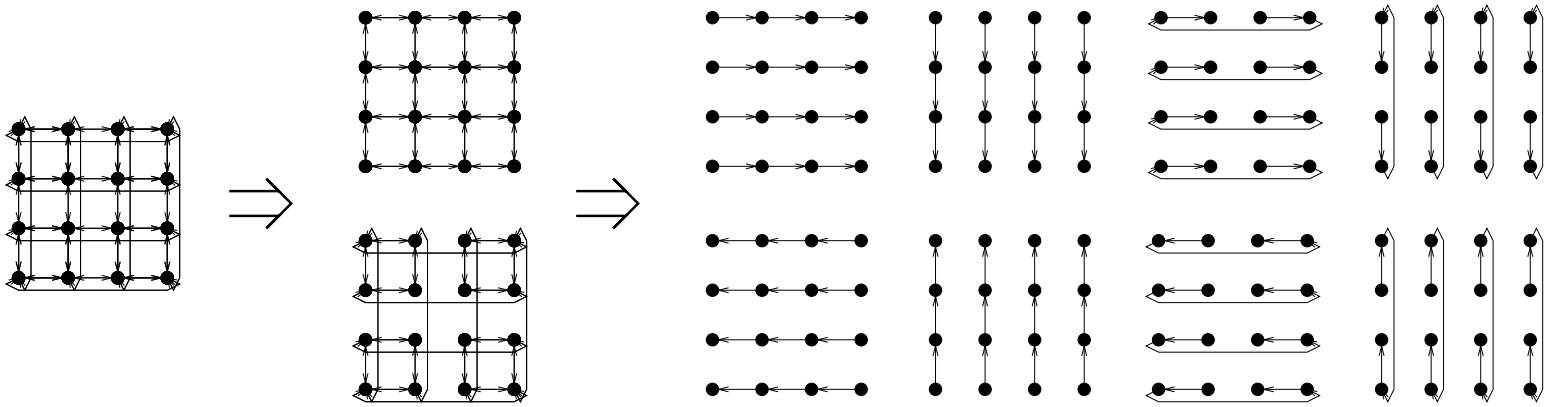
  This is achieved by multiplexing the links and supporting two separate handshakes, one for each network.

  Thus a blockage in one network cannot cause a blockage in the other.

Deadlock Freedom in Closed Networks

The same technique can be used in two or more dimensions with uni-directional or bi-directional links.

2D Bi-directional Torus



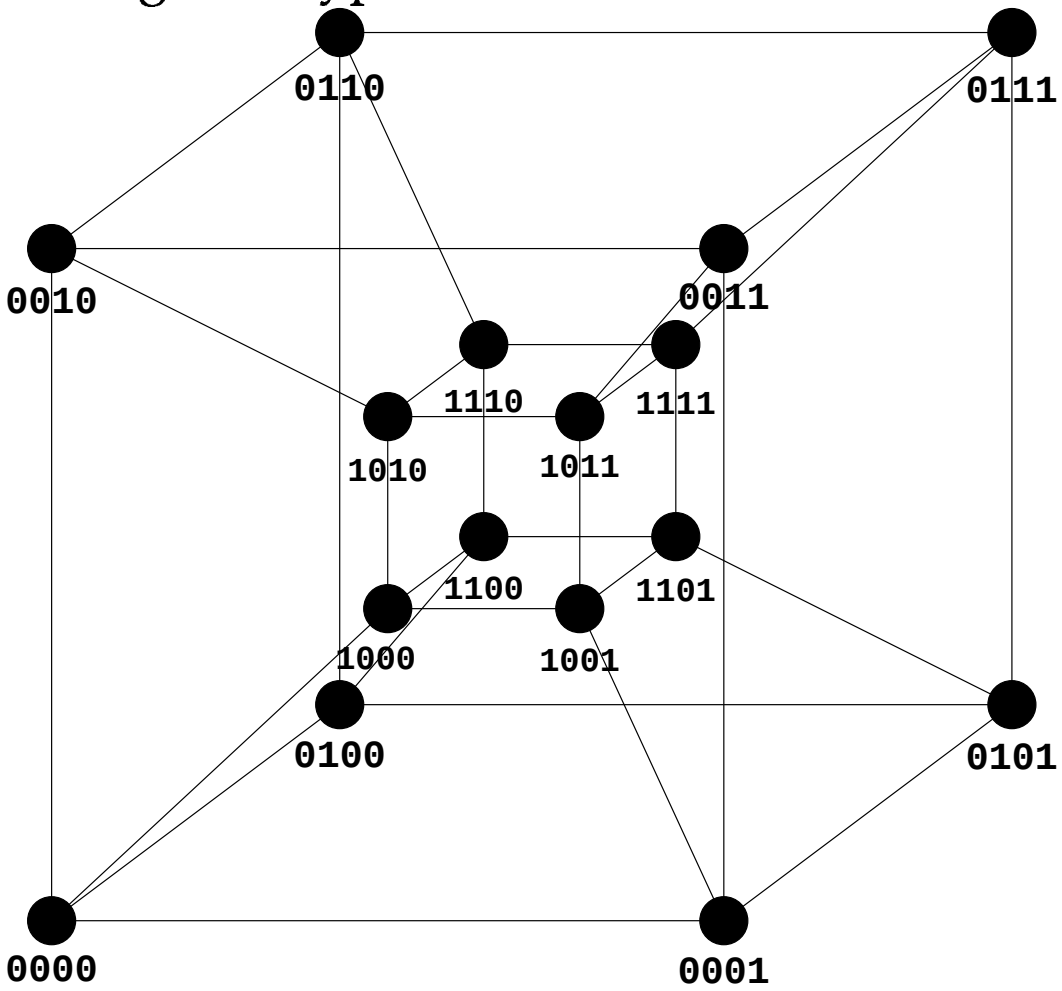
- We divide our torus into two overlapping bi-directional grids.
- Each grid can then be divided into four virtual networks.
- Again we will restrict our routing; *Route first in X and then in Y.*

Hypercube Routing

- Hypercubes can be considered as open networks or closed networks (because all PEs are edge PEs).
- An ND hypercube is divided into an ordered set of $2N$ virtual networks:
  - Dimension 1 positive
  - Dimension 1 negative
  - $\vdots$
  - Dimension N positive
  - Dimension N negative
- Routing is then performed in dimension order.
- The nodes are numbered such that the state of a single bit in the address determines whether the packet needs to be routed in a particular dimension.

Hypercube Routing

Node numbering in a hypercube network.

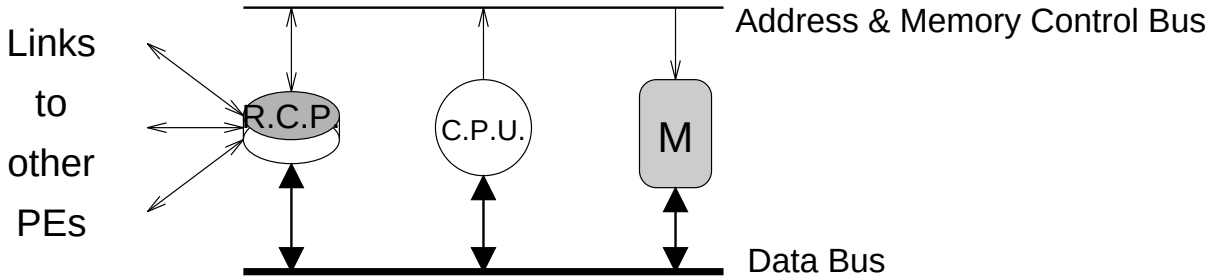


Software Routing

- With software packet routing we run a routing process on each PE.
- The routing process is timeslice multi-tasked with the other processes running on the PE.
- In a communications intensive program the PE will spend most of its time routing packets which are merely *passing through* the node.
- This will get worse as the network gets bigger.

Hardware Routing

- With hardware packet routing we provide each PE with a packet routing co-processor.
- Routing decisions can be made within one cycle thereby speeding message delivery.
- The PE's CPU need only deal with messages to or from the local node.



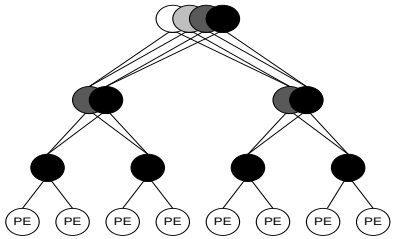
Cost/Benefit Analysis

A specialised routing co-processor may double the cost of a PE, but because it is a dedicated device it can increase performance by a greater amount.

The ideal balance is to be able to get data in and out as fast as we can process it.

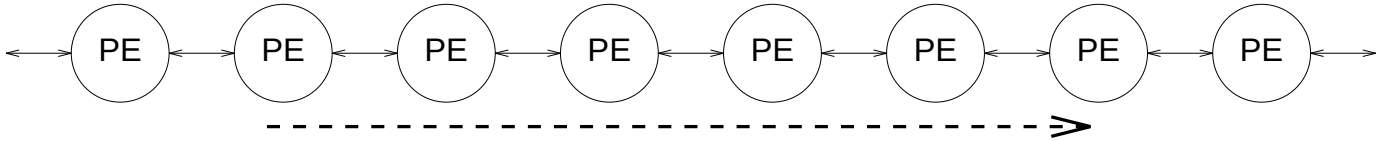
Independent Routing Processors

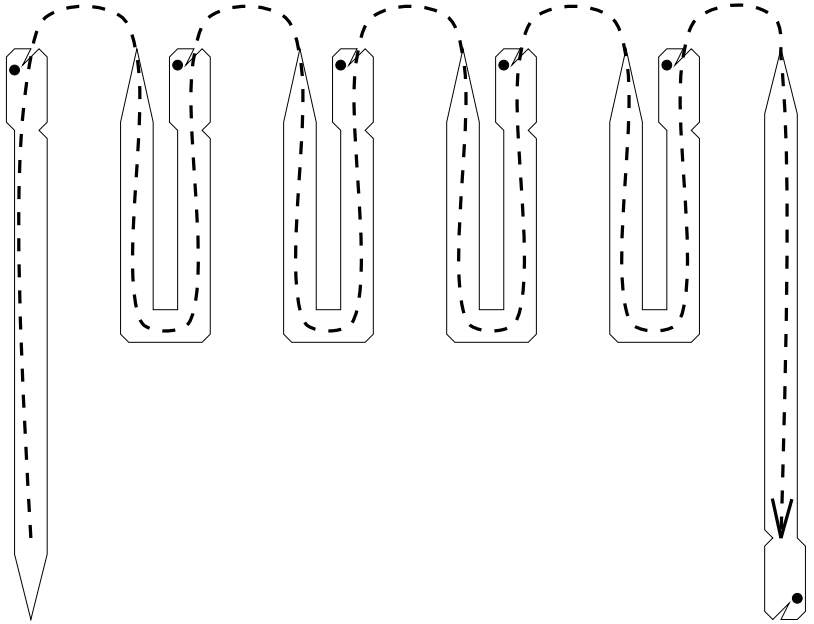
Some co-processors can exist independently of there hosts as in our *fat tree*.



Store and Forward Packet Routing

Let us examine the transport of a packet in a *store and forward* network.





Routing Performance

- The time taken for the delivery of a message is calculated as follows;

$$T = n * (l + h)/B$$

where

  - n is the number of hops.
  - l is the number of data bits.
  - h is the number of header bits.
  - B is the link bandwidth in bits per second.

Assumptions

  - There is no overhead for decision making (hardware routing).
  - We have an empty network (packets will be further delayed in a busy network).

Alternative Routing Strategies

We are waiting for a time $t = (l+h)/B$ at each node while the whole packet is stored before making a routing decision.

Yet we see that after a time $t' = h/B$ we have received the header information. We have all the information required in order to make our routing decision.

If we make our routing decision at t', we gain in two ways:

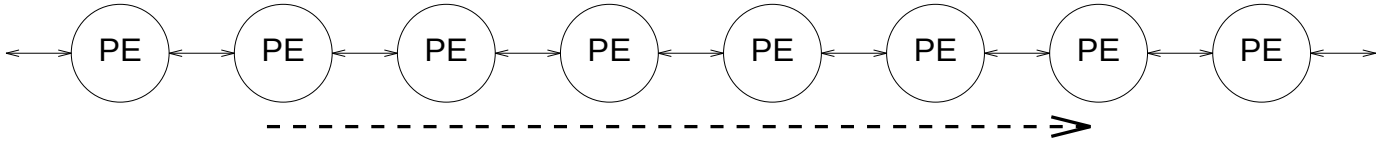
- We reduce the *message latency*.
- We no longer require large buffers at each node.

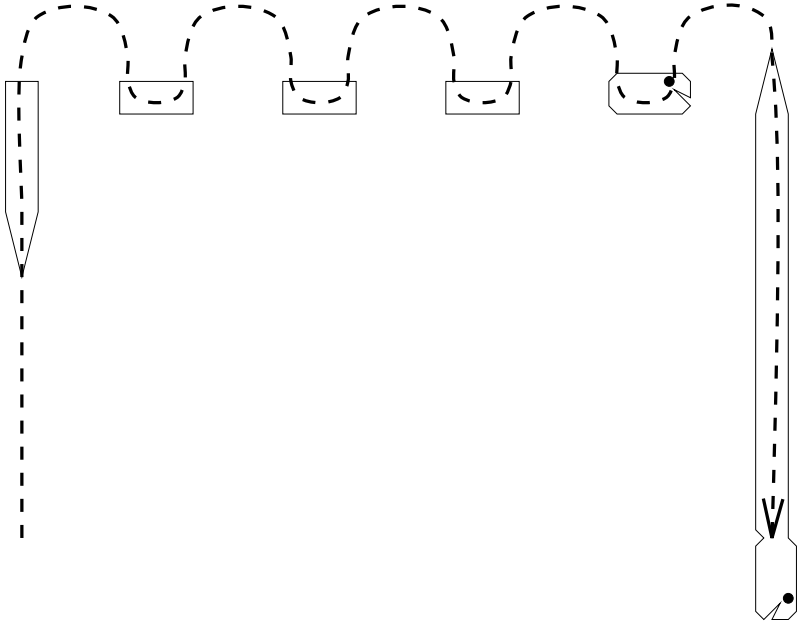
  The buffer size is determined by the header size. We can return to message routing rather than packet routing (if we so desire).

We have *Wormhole Message Routing*

Wormhole Message Routing

Let us examine the transport of a packet in a *wormhole* network.





Routing Performance

- The time taken for the delivery of a message is calculated as follows;

$$T = (l + (n * h))/B$$

where

  - n is the number of hops.
  - l is the number of data bits.
  - h is the number of header bits.
  - B is the link bandwidth in bits per second.

c.f.

$T = n * (l + h)/B$ for Store & Forward

Again we are assuming hardware routing in an empty network.

The Problem with Wormhole

- We have looked at the performance of *wormhole* routing in an empty network and found it considerably better than *store & forward*.
- One of the characteristics of *wormhole* routing is that messages are spread out over the network. When a single message is delayed due to a temporary blockage that single message will remain spread out across the network, thus potentially blocking a large number of other messages.
- In this way it appears that *store & forward* routing may have advantages in busy networks, as a single waiting packet can only block one node.

Virtual Cut-Though Packet/Message Routing

Virtual cut-through packet routing is a hybrid of *wormhole* and *store & forward*.

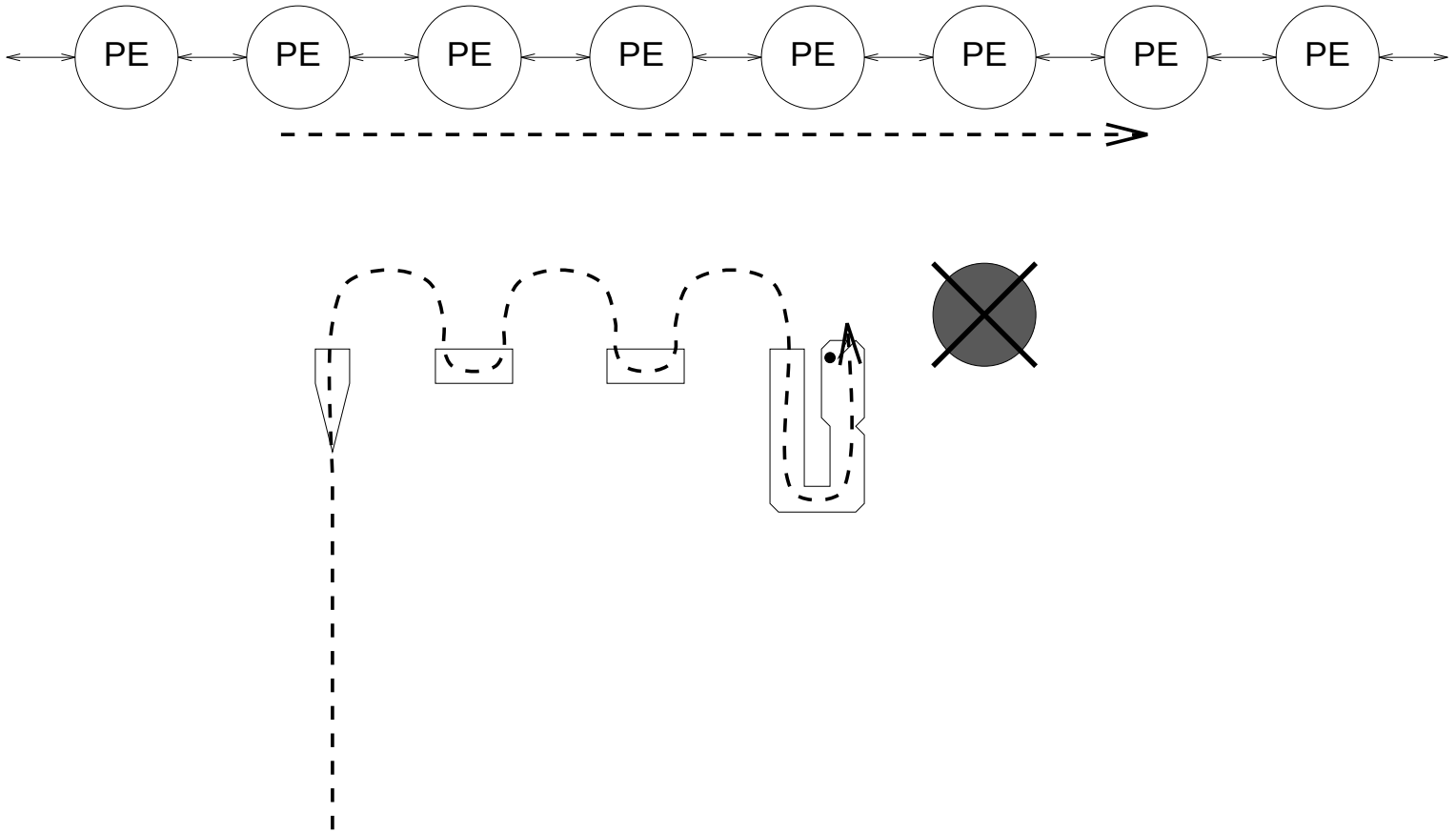
- In an empty network it behaves like wormhole routing where each packet spreads itself across the network.
- When a packet encounters a blockage, the head stops at the blockage while the tail continues until all of the packet is buffered at a single node.

  When the blockage is gone the packet will again spread out across the network.
- In this way we get the best of both worlds.

We can implement either packet routing or message routing with *virtual cut-through*. If we implement message routing then a long message may remain spread out over several nodes when its progress is blocked.

Virtual Cut-Though Packet/Message Routing

Buffering of a blocked packet in *virtual cut-through* routing:



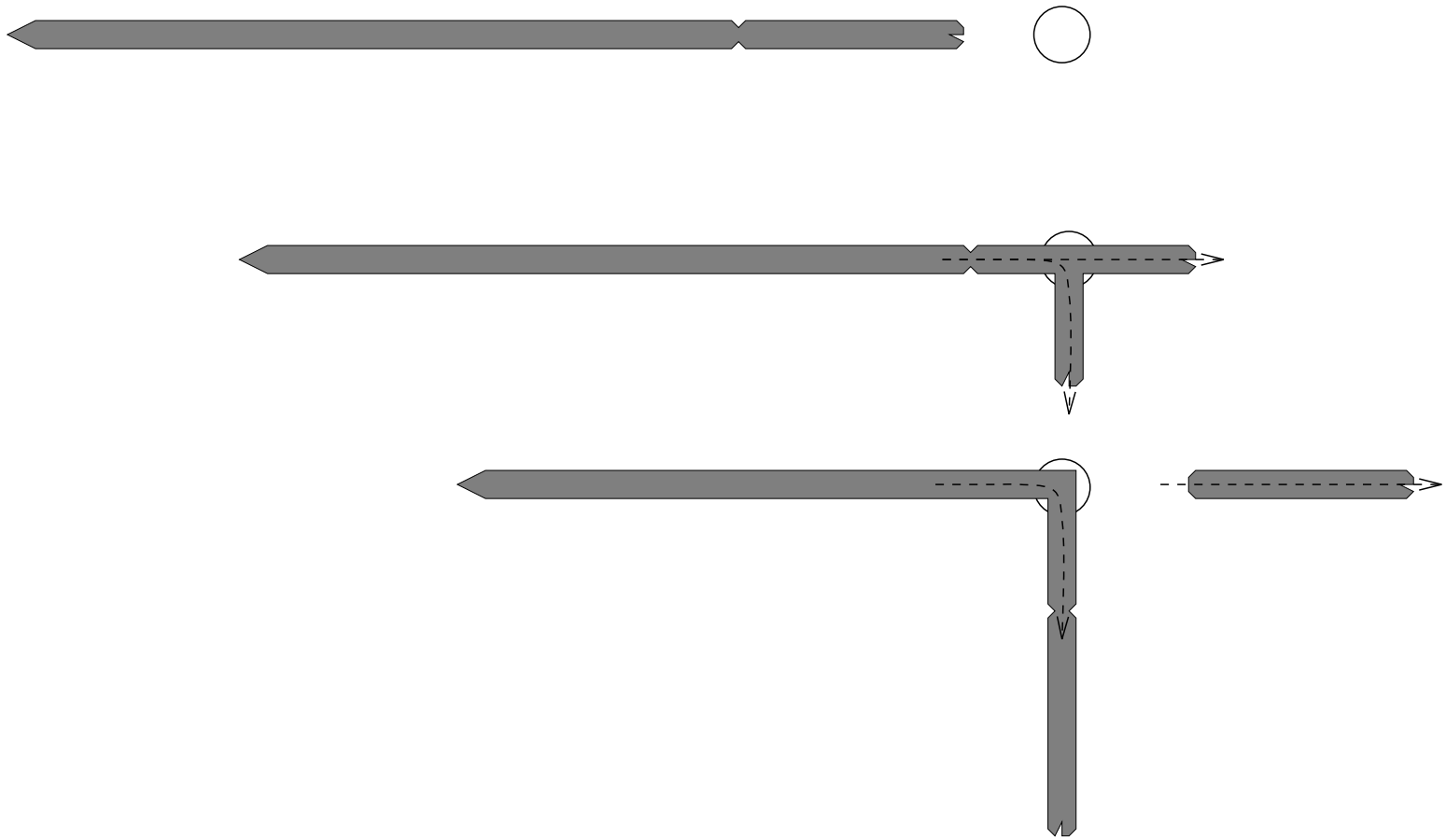
Mad Postman Packet/Message Routing

- The Leading Edge

- Mad Postman Routing takes *virtual cut-through* one stage further.
- Instead of delaying the message until the header can be checked, the *mad postman* passes the received information immediately to the next processor while taking a copy for itself. The minimum delay of one cycle is applied at each node.
- When the header has been received (& sent on), a decision on the routing of the data can be made.
- In the event of a temporary blockage a full packet can be buffered at a single node (c.f. virtual cut-through).

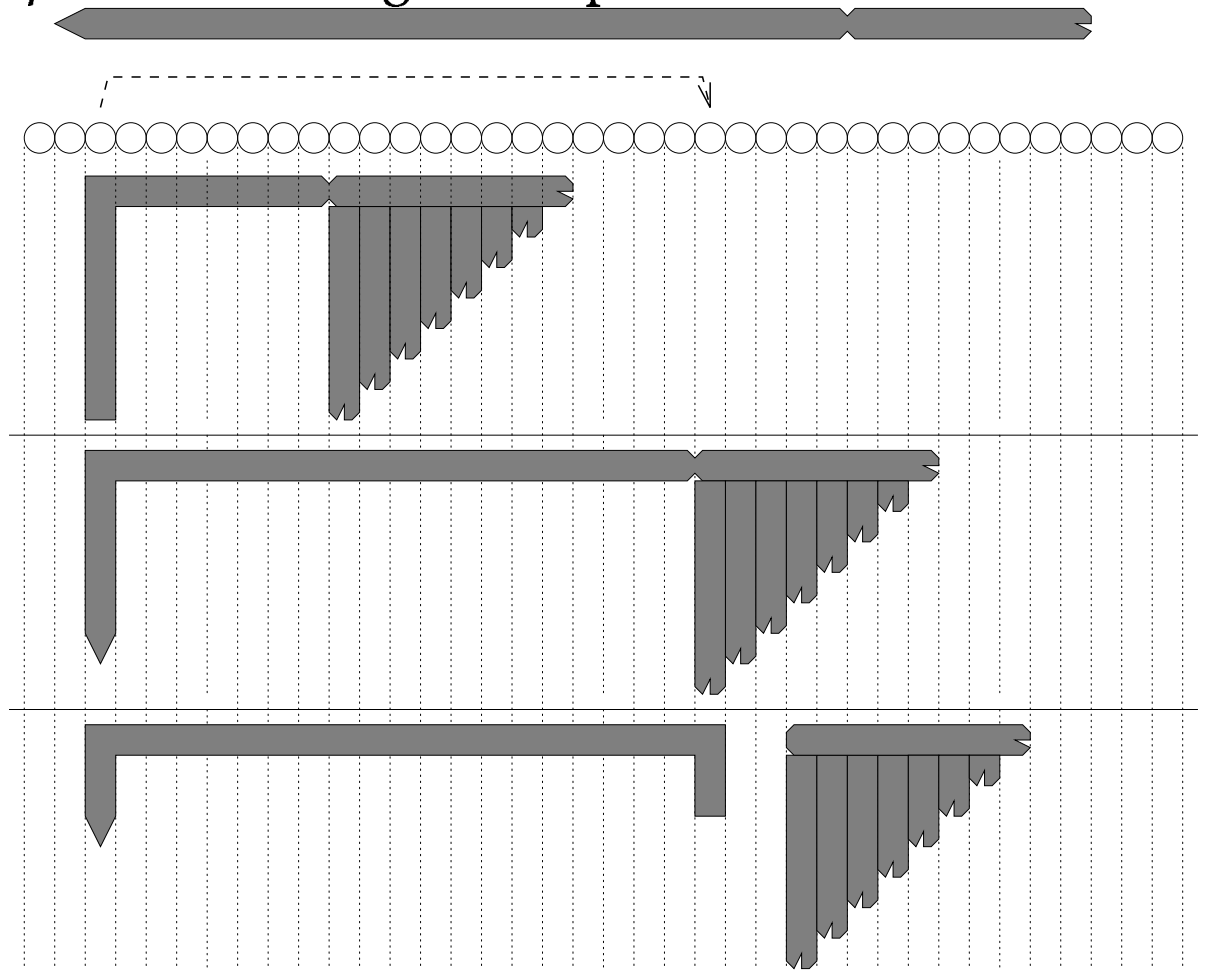
Mad Postman Packet/Message Routing

1D *Mad postman* routing - Individual behaviour :



Mad Postman Packet/Message Routing

1D *Mad postman* routing - Group behaviour :



Routing Performance

- The time taken for the delivery of a message is calculated as follows;

$$T = (l + h + n - 1)/B$$

where

  - n is the number of hops.
  - l is the number of data bits.
  - h is the number of header bits.
  - B is the link bandwidth in bits per second.

c.f.

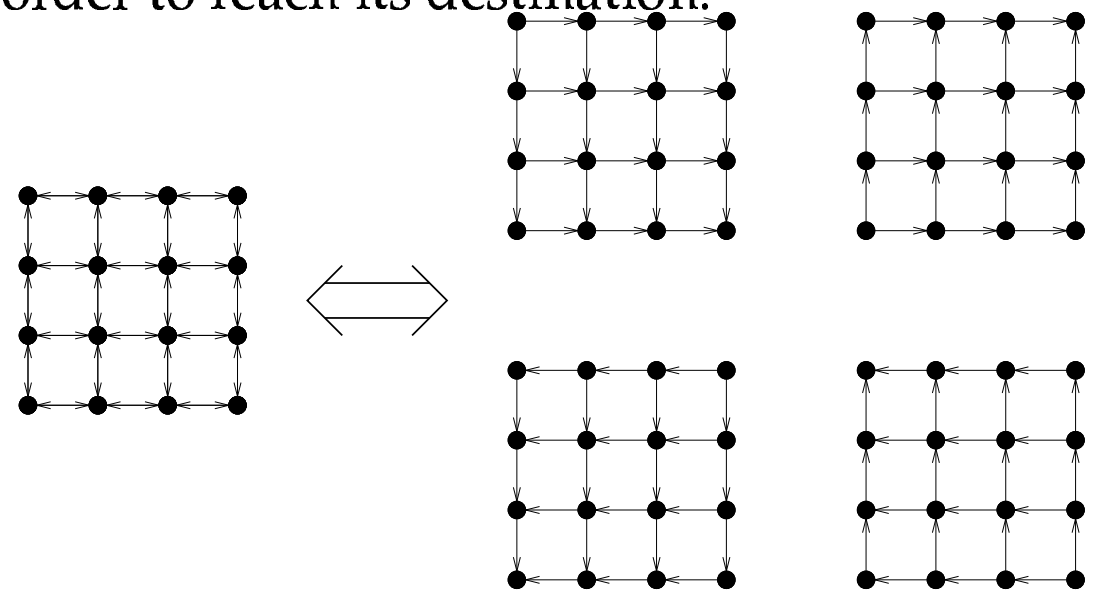
$T = n * (l + h)/B$ for Store & Forward.

$T = (l + (n * h))/B$ for Wormhole and Virtual Cut-Though.

2D Mad Postman

The present implementation of *mad postman* routing is 2D grid split into four virtual networks.

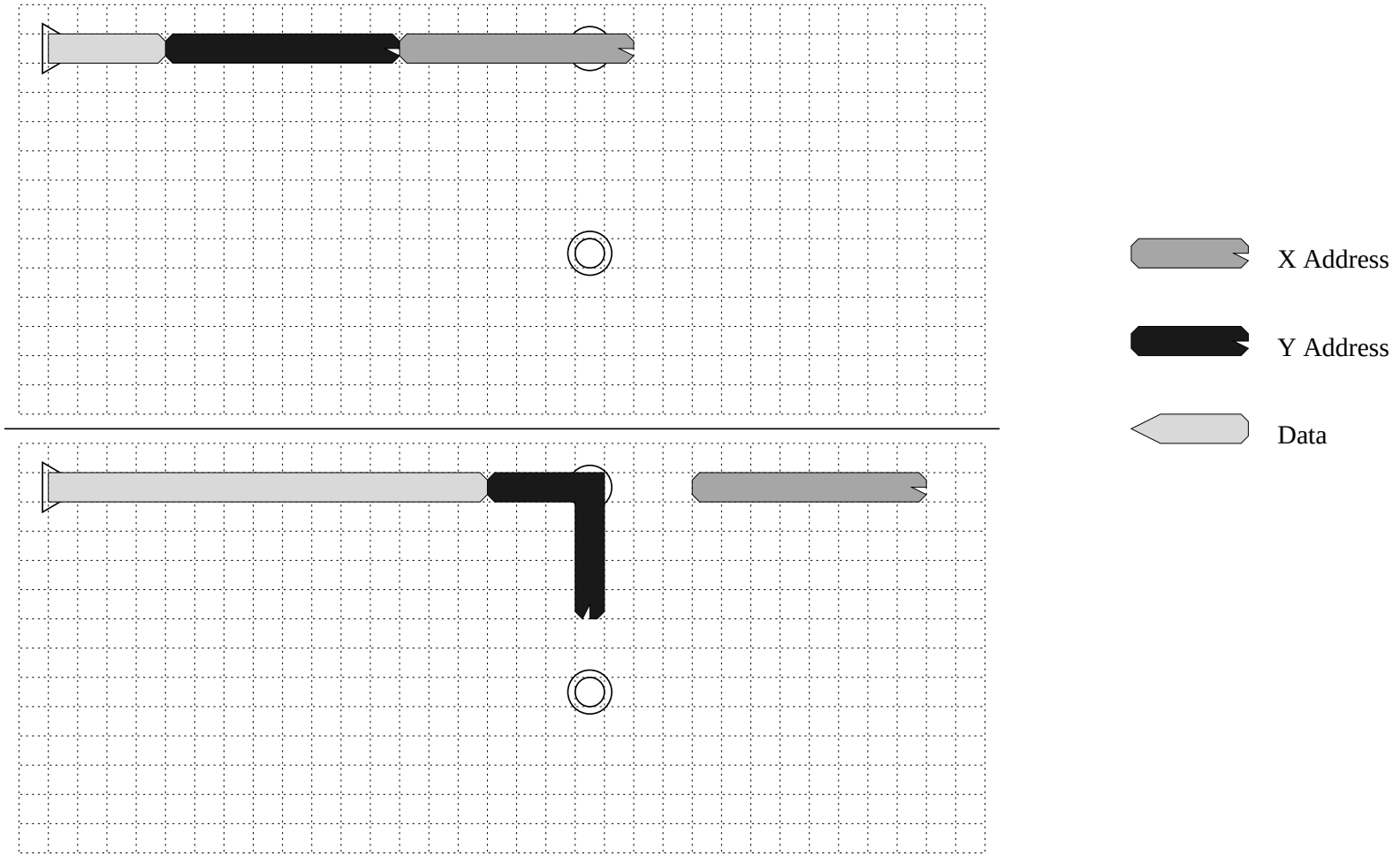
The $\{+X, +Y\}$, $\{+X, -Y\}$, $\{-X, +Y\}$ & $\{-X, -Y\}$ allow for adaption without deadlock. A message need travel in only one virtual network in order to reach its destination.



The *mad postman* provides independent links for the different networks.

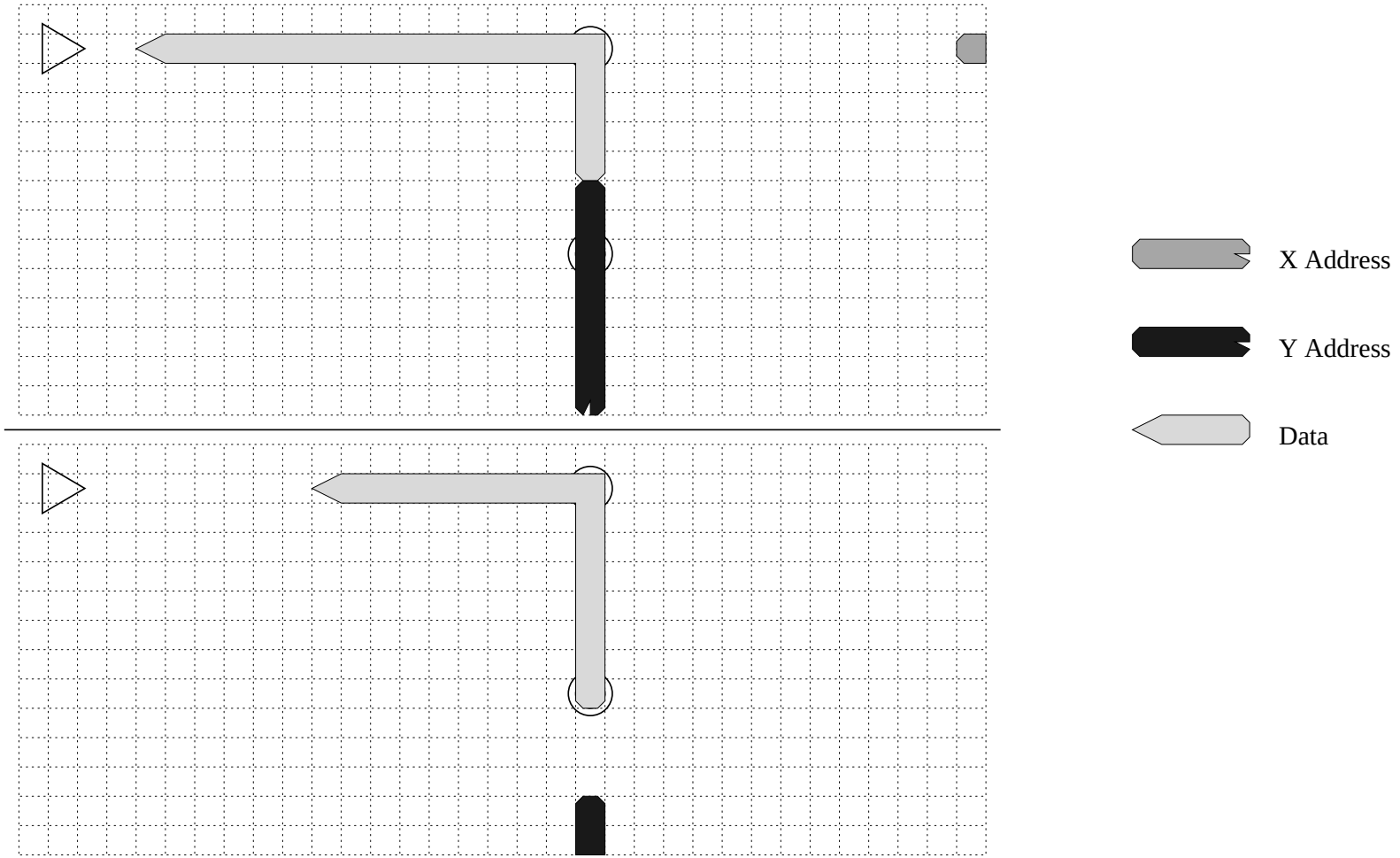
2D Mad Postman

A) Arrival in X



2D Mad Postman

B) Arrival in Y



2D Mad Postman with Adaption

- When a packet is blocked at a node it can be fully buffered at that node (c.f. virtual cut-through).

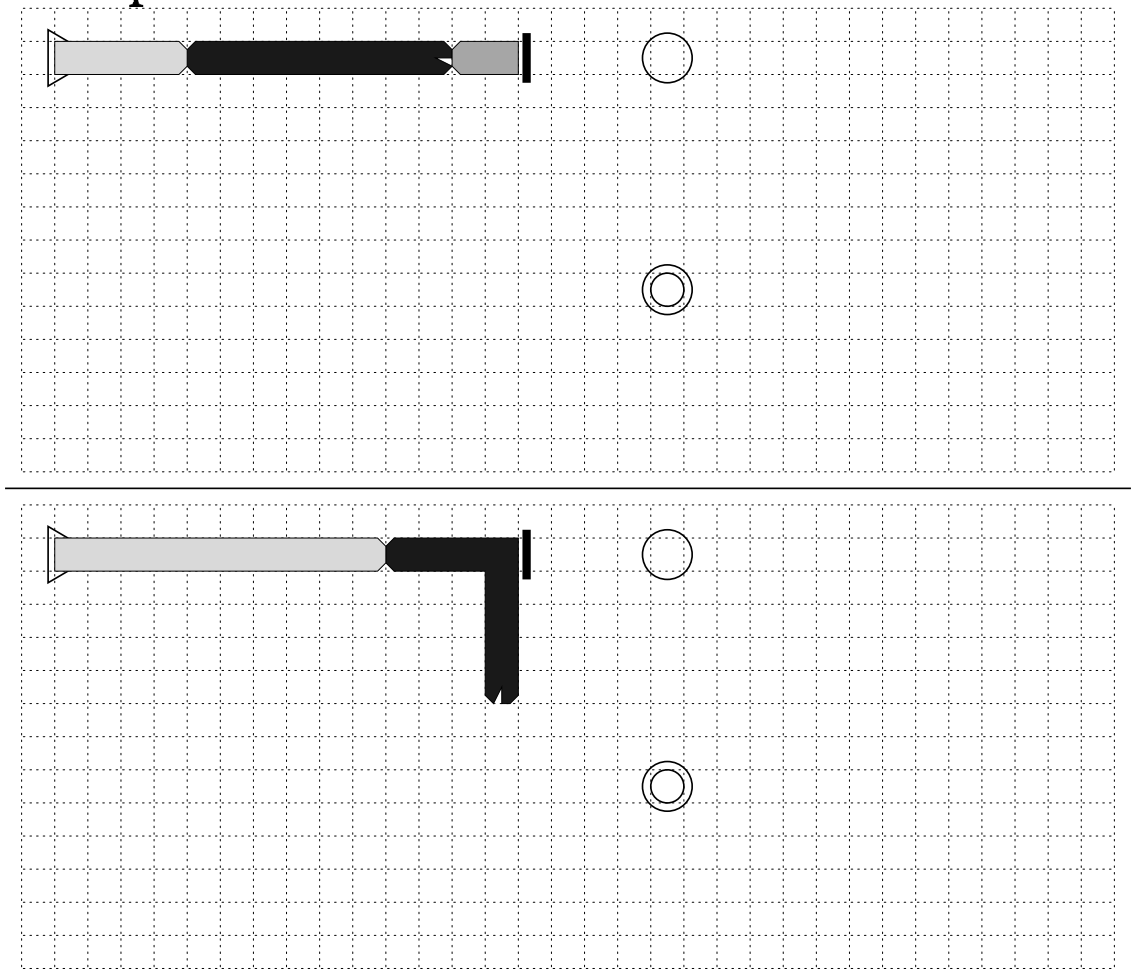
- If after the first header has been buffered there is an unblocked channel in the other dimension, then the packet will be re-directed into that dimension.

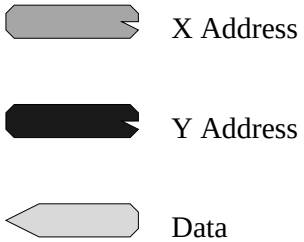
- For this to happen the headers must be swapped such that the leading header corresponds to the direction of travel.

- As a result of this initial delay for header swapping the node where the blockage occurred will continue to buffer a header sized chunk of the packet.

2D Mad Postman with Adaption

B) Adaption X to Y





2D Mad Postman with Adaption

B) Arrival in Y