Modeling HexNet in ACL2

Developing a formal specification for HexNet

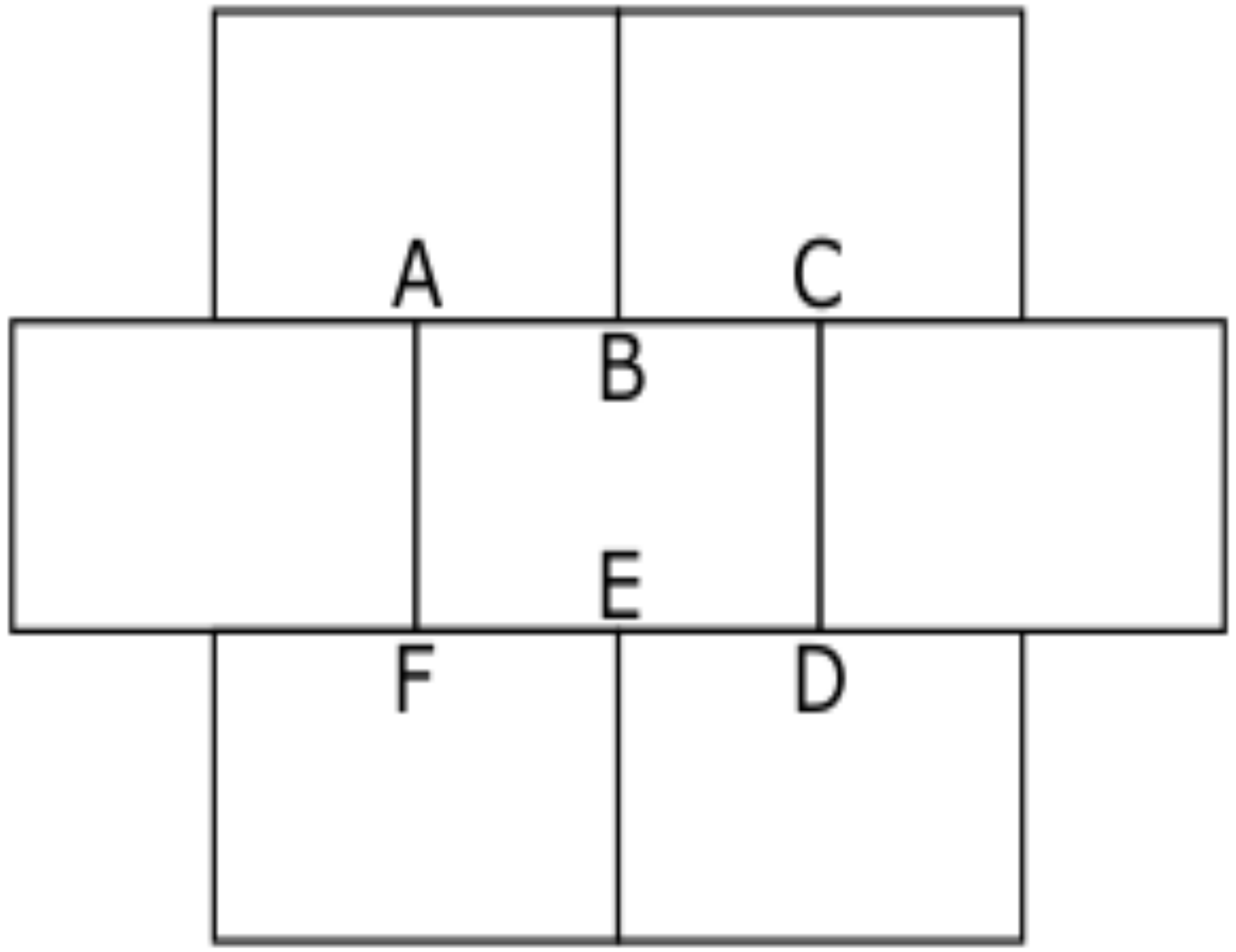
Presentation by Ebele Esimai

Outline

- Overview of HexNet and its features
- Current Model in ACL2
- Desired theorems
- Future work
- Some ACL2 functions

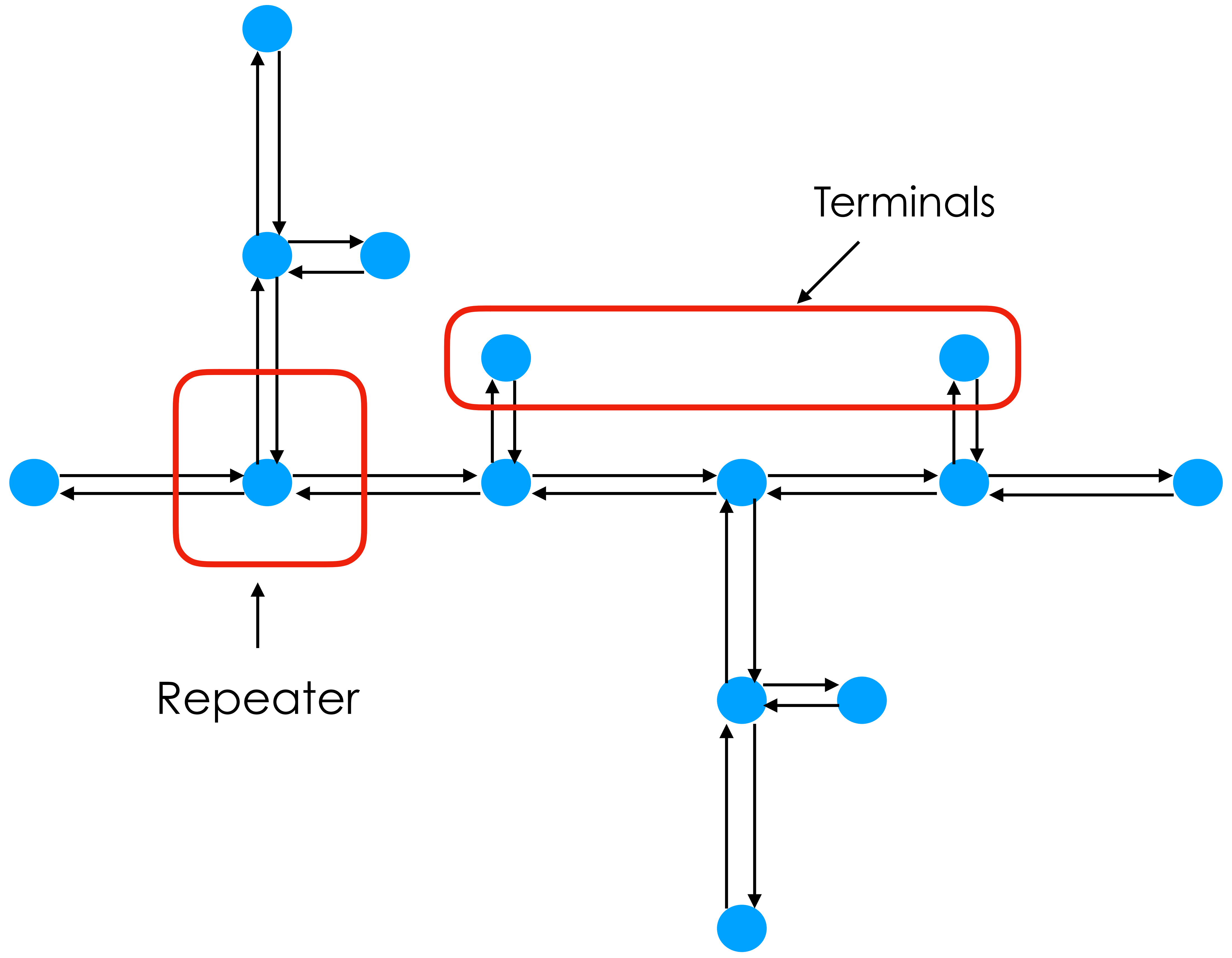
HexNet Layout

- Hexagonal layout of the network (proposed by Ivan Sutherland 2013)
- Each node in the network is a called a JUNCTION
  - arbitration
  - packet routing
  - packet update
- Each edge is a link, storage element for packets
- Branching and Merging in the network is two-way



Junctions

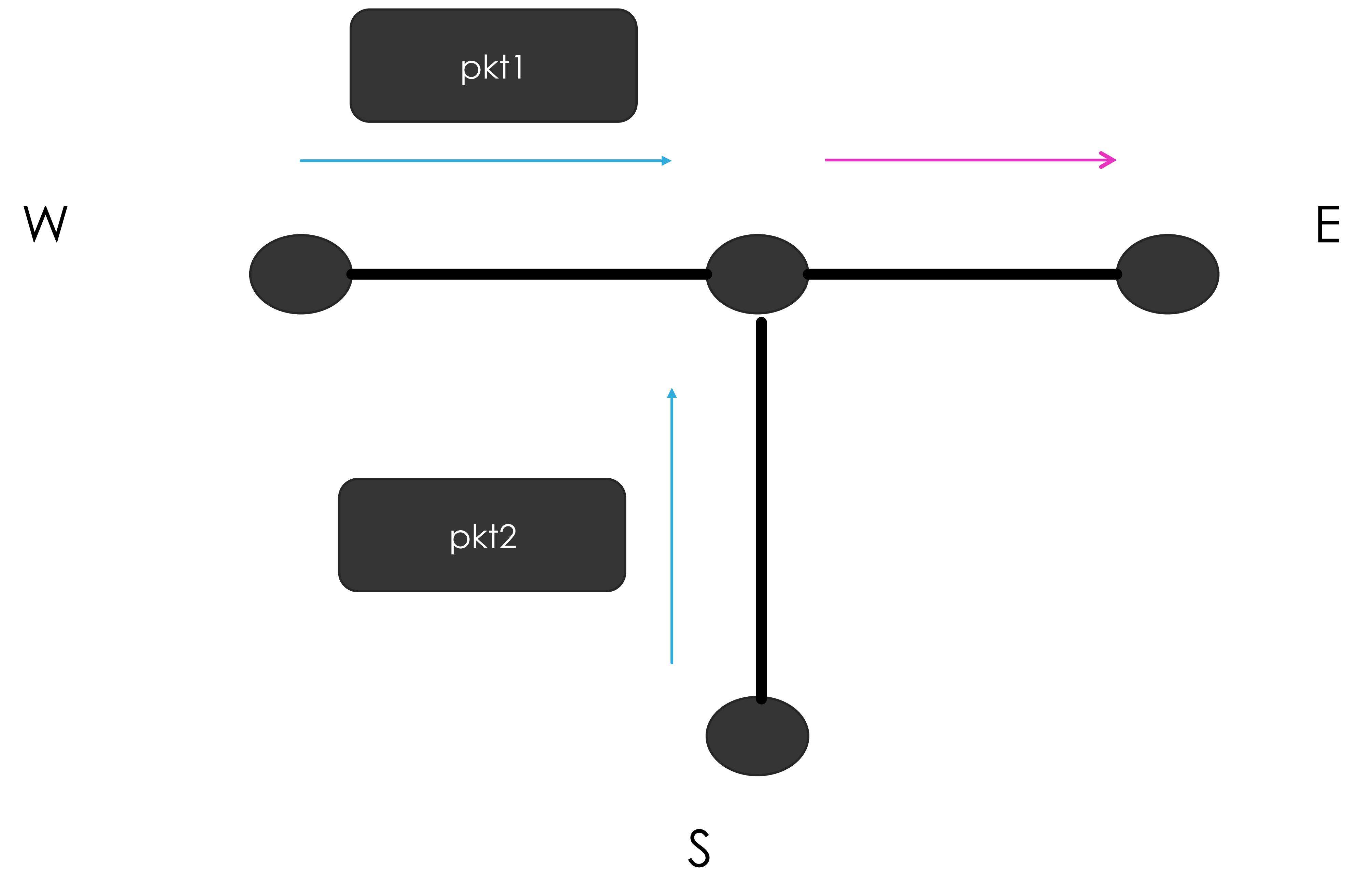
- Junctions can be of two types
  - Terminal : This is the point of packet entry or exit from the network.
  - Repeaters : This accepts packets and forwards it to another junction.



Features of HexNet

Arbitration

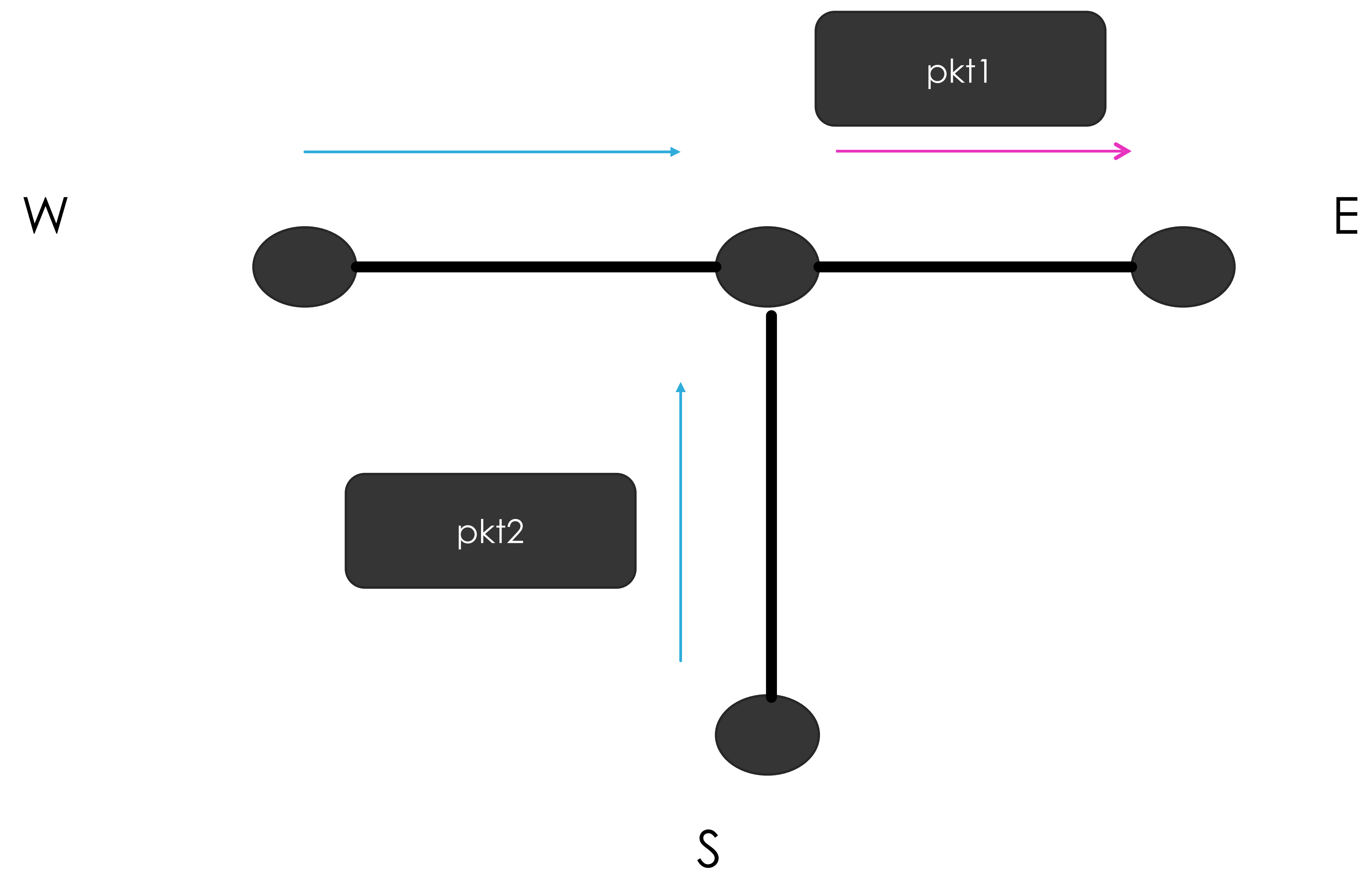
- Correct decision behavior of the network when packets arrive on two input links at nearly the same time.
- A mechanism for fairness is necessary. The model stores the preference from the arbitration decision at the previous cycle.
- Arbitration is based on the output direction.
- One of the packets is chosen to proceed while the second waits.
- In the following cycle, the packet waiting gets to proceed.



Features of HexNet

Arbitration

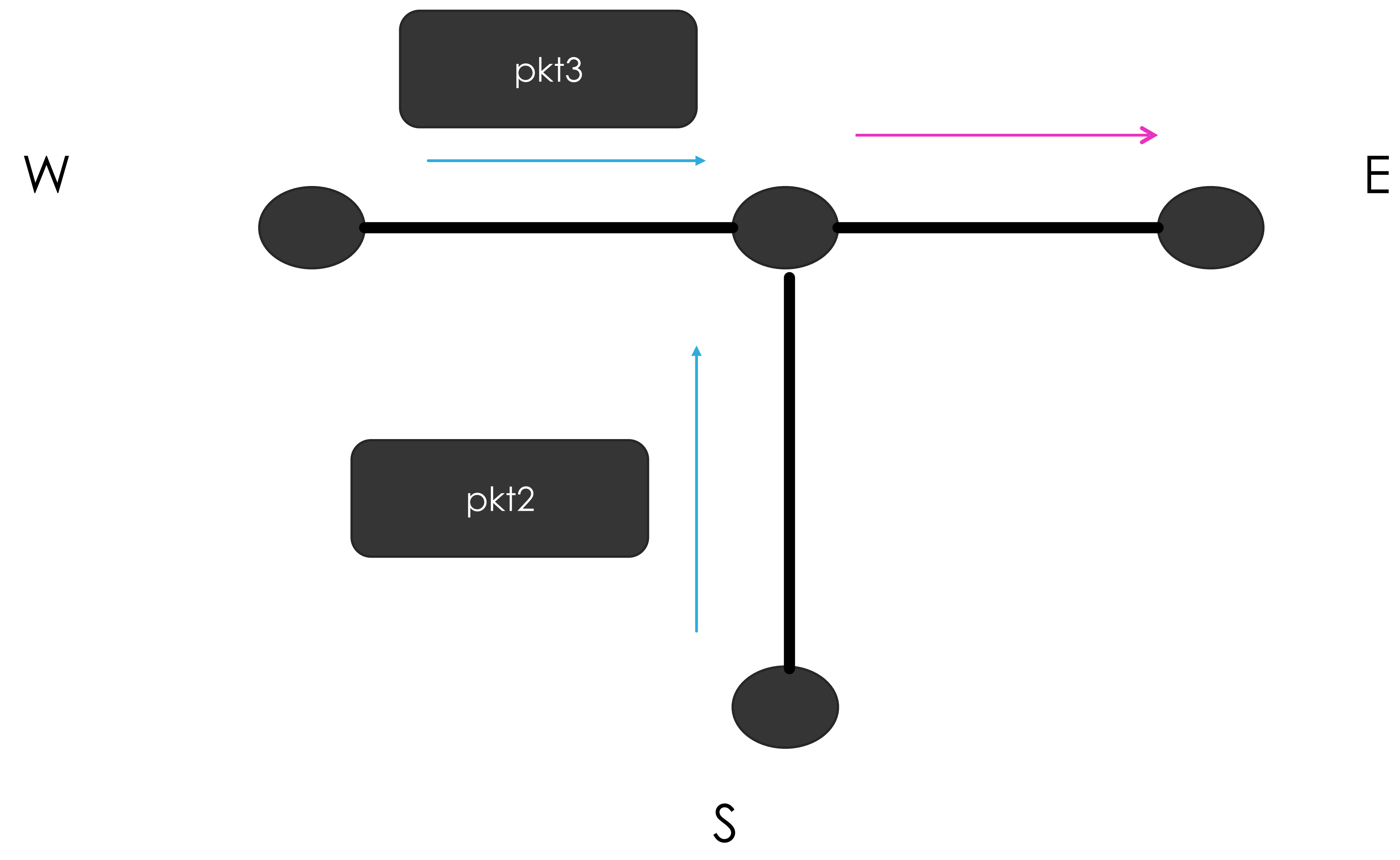
- Correct decision behavior of the network when packets arrive on two input links at nearly the same time.
- A mechanism for fairness is necessary. The model stores the preference from the arbitration decision at the previous cycle.
- Arbitration is based on the output direction.
- One of the packets is chosen to proceed while the second waits.
- In the following cycle, the packet waiting gets to proceed.



Features of HexNet

Arbitration

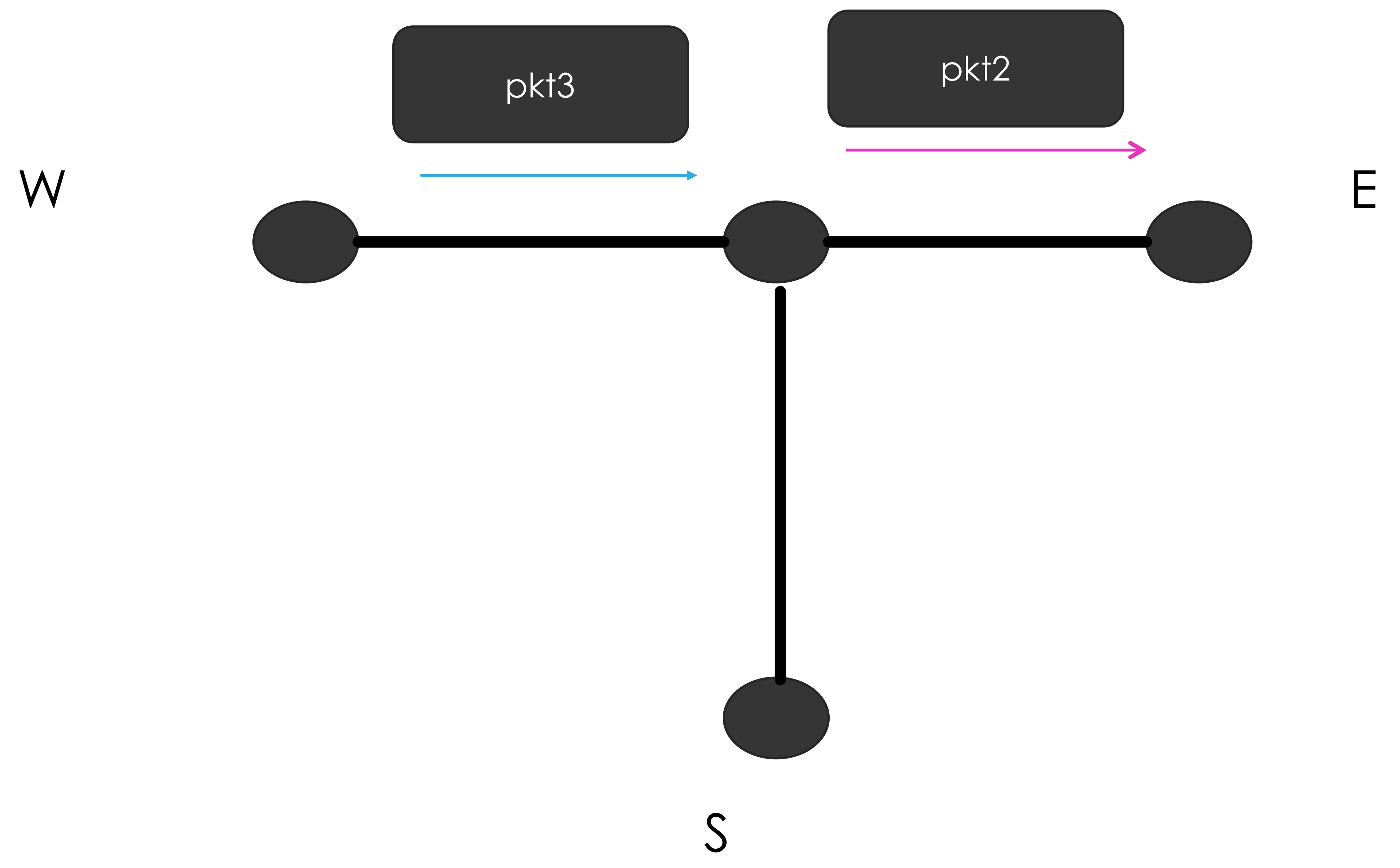
- Correct decision behavior of the network when packets arrive on two input links at nearly the same time.
- A mechanism for fairness is necessary. The model stores the preference from the arbitration decision at the previous cycle.
- Arbitration is based on the output direction.
- One of the packets is chosen to proceed while the second waits.
- In the following cycle, the packet waiting gets to proceed.



Features of HexNet

Arbitration

- Correct decision behavior of the network when packets arrive on two input links at nearly the same time.
- A mechanism for fairness is necessary. The model stores the preference from the arbitration decision at the previous cycle.
- Arbitration is based on the output direction.
- One of the packets is chosen to proceed while the second waits.
- In the following cycle, the packet waiting gets to proceed.



Features of HexNet

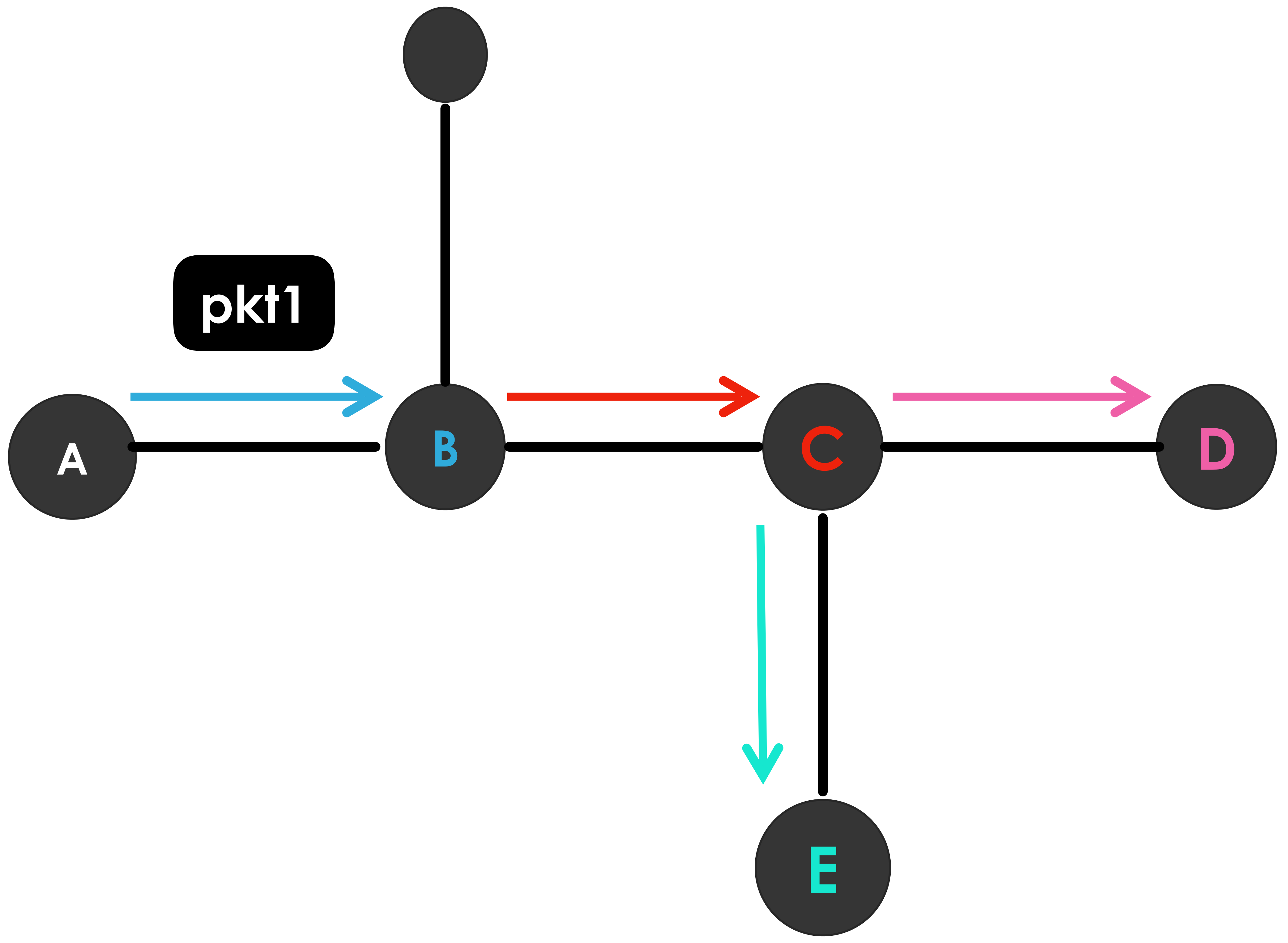
Packet Routing

- The network uses **Advanced Address Decoding**
- This means that the routing function calculates the next direction, i.e. turn-signal, a step in advance. This means before it actually takes a turn.
- The routing function decides the packet direction by comparing the final destination of the packet and the address of next junction.
- Let the current turn_signal of pkt1 be East, thus, the packet would continue to junction C.

Turn_signal (pkt1) = East

- But before leaving junction B, it calculates the value of the next turn_signal, that is, the direction it should take when it leaves junction C.

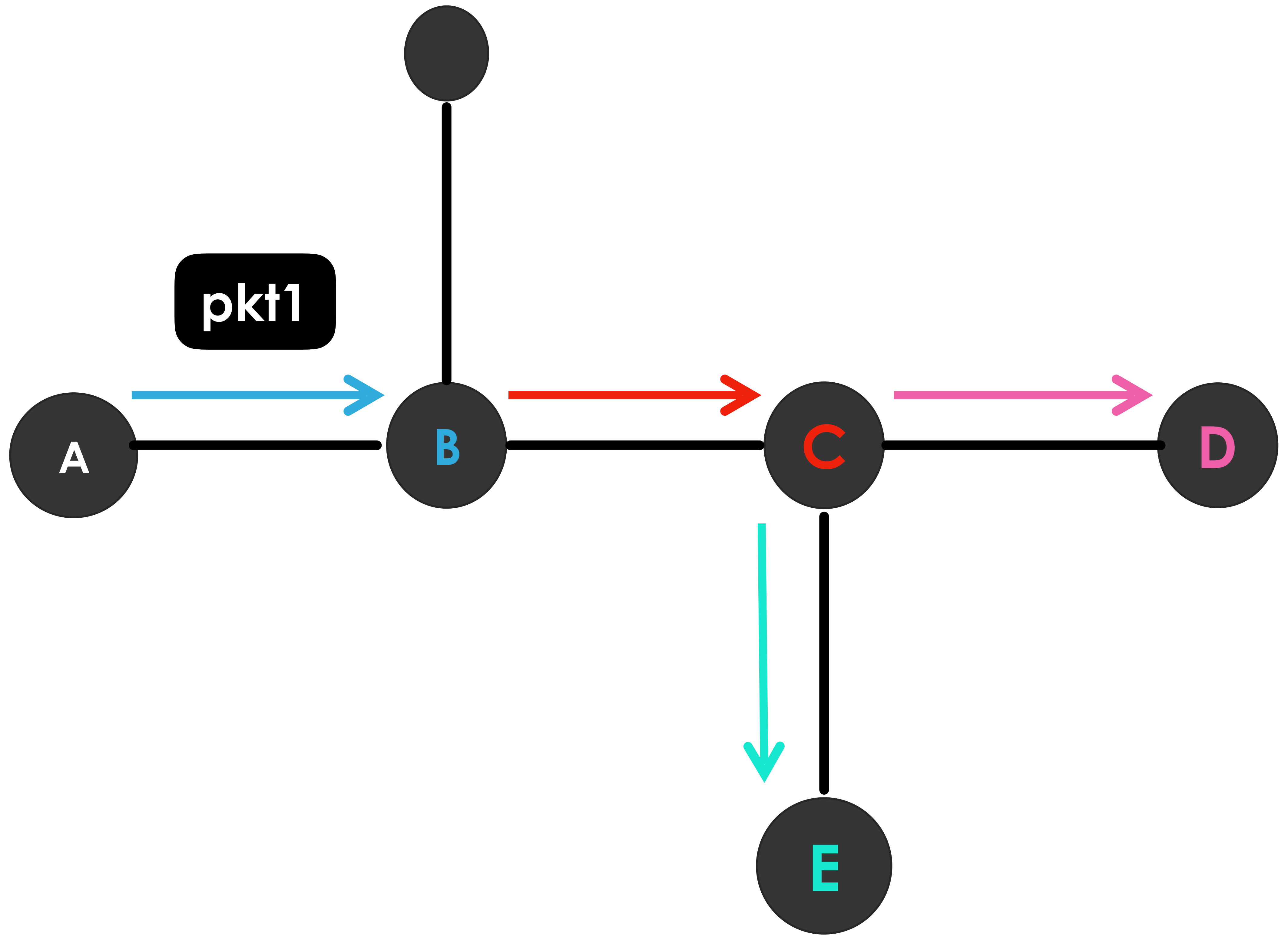
(compare_addresses junctC final_destination) => East or South



Packet Routing

Routing Algorithm

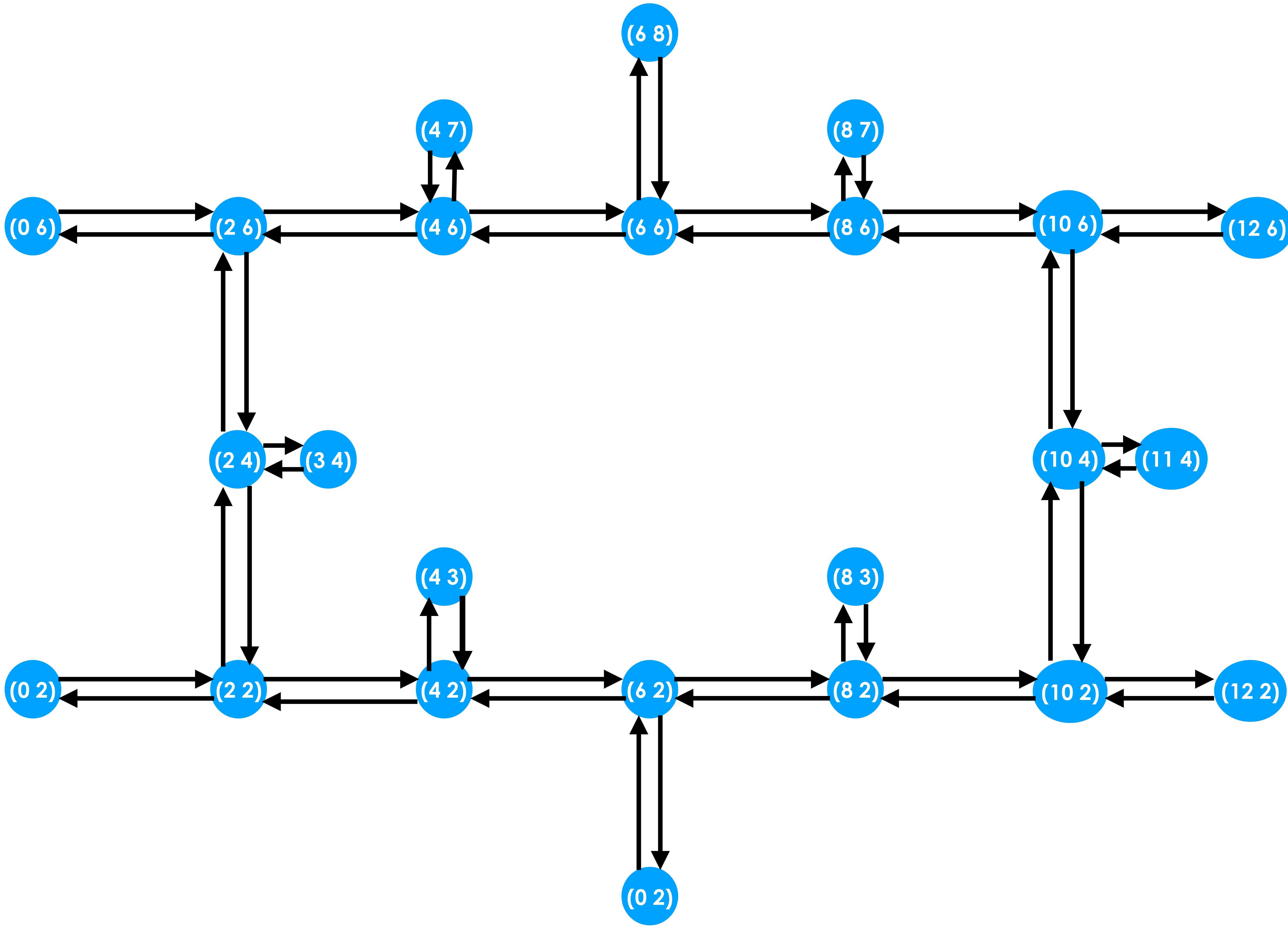
- Get the next possible junctions from current junction.
- Check if either junction is the final destination address
- If not, check if either junction is a terminal. If one is a terminal, return the direction of the repeater.
- If either junction is a repeater, compare the current junction address with the final destination address.
  - return the direction of repeater which is closest to the final destination
  - vertical movement is preferred



- Next possible junctions from **C** are **D** and **E**

Current Model in ACL2

- The network is modeled as a graph.
- Addressing is based on a cartesian grid, as such, each junction is list of its x coordinate and its y coordinate
- Packet movement in the network is bidirectional.
- Packet entry and exit from the network is through the terminals



Packets

Pkt1

‘(E (8 7) (0 2) ‘data)

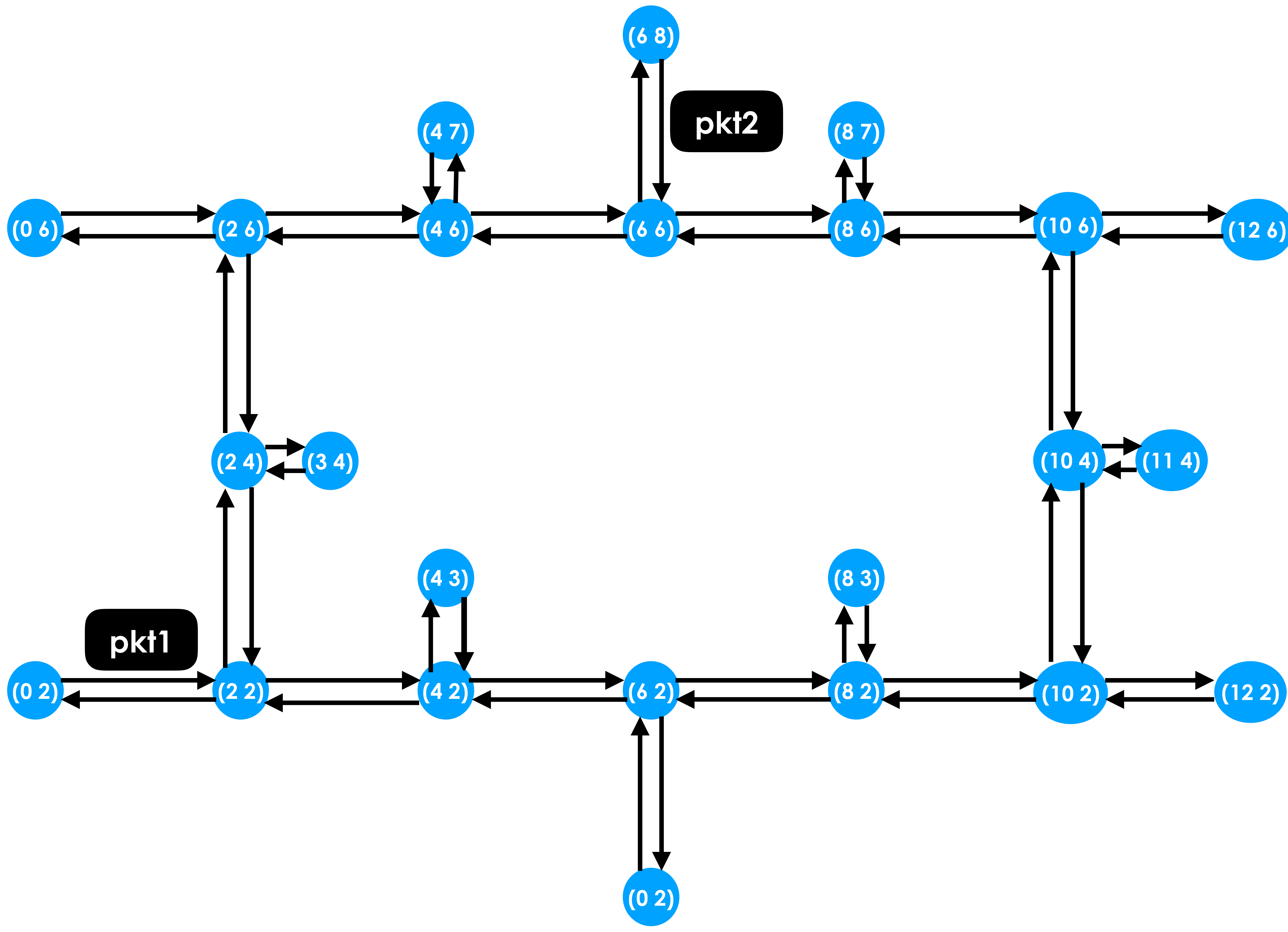
- A packet is modeled as a list that contains
  - A turn signal :- the next direction the packet should take
  - Final terminal :- the address of final destination of the packet
  - Source terminal :- the address of the origin of the packet
  - The information to be passed along

- The packets are stored on the links.
- A link-table shows the current state of all the links in the network at a given time.
- A link can either be empty or full.
- In the link-table, each link is depicted as either empty or with the packet stored on it.

```
(defconst *linktable*
      '((S2 (E (12 2) (0 2) data1))
        (S1)
        (S4)
        (S3)
        …
       (T19 (E (12 2) (6 8) data2))
        …))
```

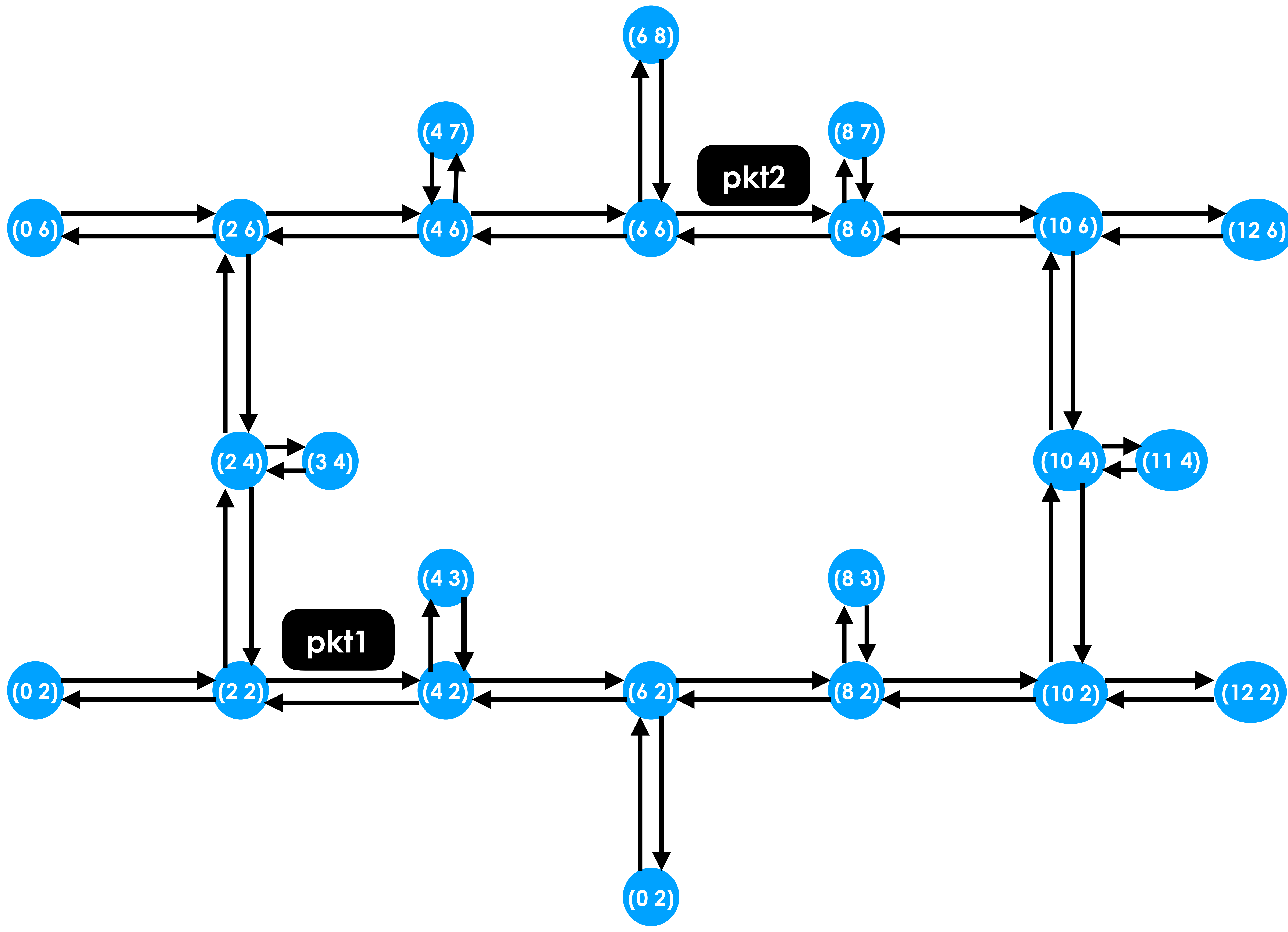
Packet Movement

- A packet can move along its path to its final destination as long as
  - the packet is well-formed
  - the receiving link is empty
  - if in a tie with another packet, the packet is picked by the arbiter
  - the routing function gives a valid direction
- Here, the system checks if the receiving link is empty. If full, no update is made.
- Otherwise, if the packet is valid and it is picked by the arbiter, then it calculates it next turn_signal with the routing function and repacks the packet with the new turn_signal.
- Then it updates the receiving link with the new packet and removes the old packet from the sending link.



Packet Movement

- A packet can move along its path to its final destination as long as
    - the packet is well-formed
    - the receiving link is empty
    - if in a tie with another packet, the packet is picked by the arbiter
    - the routing function gives a valid direction
- Here, the system checks if the receiving link is empty. If full, no update is made.
- Otherwise, if the packet is valid and it is picked by the arbiter, then it calculates it next turn_signal with the routing function and repacks the packet with the new turn_signal.
- Then it updates the receiving link with the new packet and removes the old packet from the sending link.



Adding new packets to the network

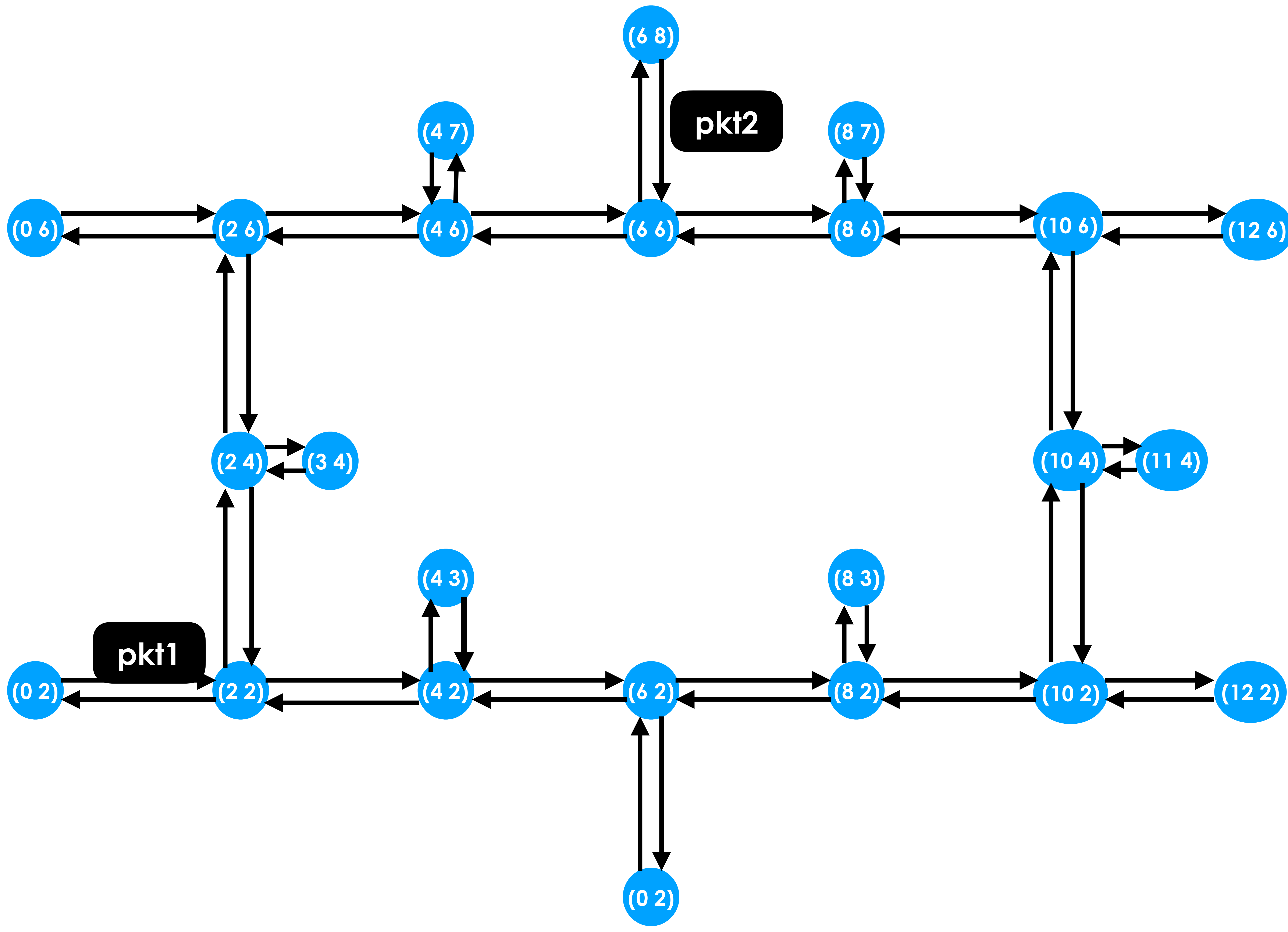
- A stream of inputs with some packet information - a final destination address, a source address and information to be passed along
- Calculate the first direction of the packet using the routing algorithm.
- Get the link name, where the packet will stored upon entry into the network
- If the link is not full, fill the link
- For example,

input stream = (list (list '((8 7) (3 4) data)

'((6 8) (11 4) data)))

- The new link table would be

((S4 (E (12 2) (0 2) DATA))	<= pkt1
(S15 (N (8 7) (3 4) DATA))	<= pkt3
(S17 (N (6 8) (11 4) DATA))	<= pkt4
(S26 (E (12 2) (6 8) DATA))...)	<= pkt 2



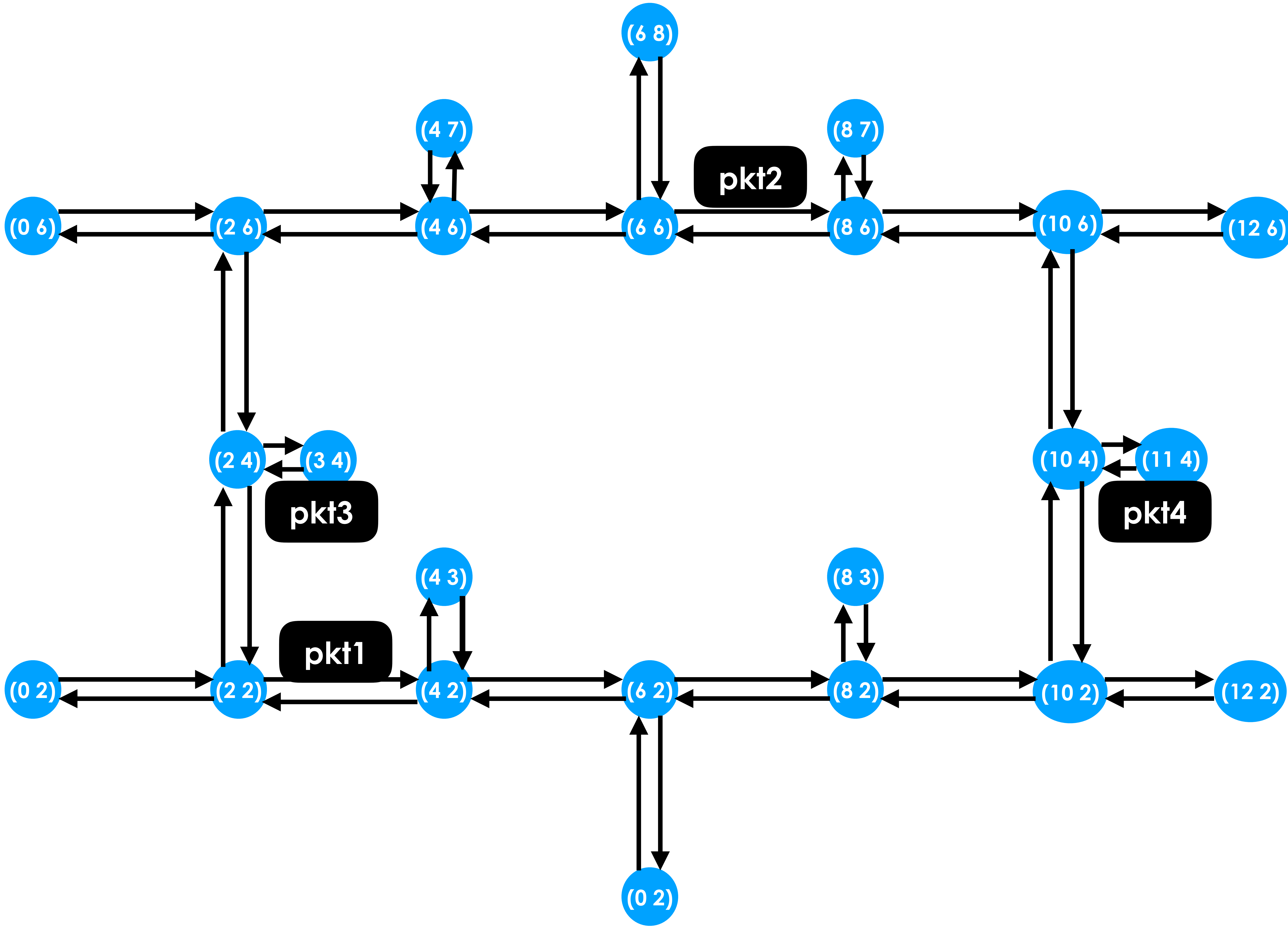
Adding new packets to the network

- A stream of inputs with some packet information - a final destination address, a source address and information to be passed along
- Calculate the first direction of the packet using the routing algorithm.
- Get the link name, where the packet will stored upon entry into the network
- If the link is not full, fill the link
- For example,

input stream = (list (list '((8 7) (3 4) data)
'((6 8) (11 4) data)))

- The new link table would be

((S4 (E (12 2) (0 2) DATA)) <= pkt1
(S15 (N (8 7) (3 4) DATA)) <= pkt3
(S17 (N (6 8) (11 4) DATA)) <= pkt4
(S26 (E (12 2) (6 8) DATA))...) <= pkt 2



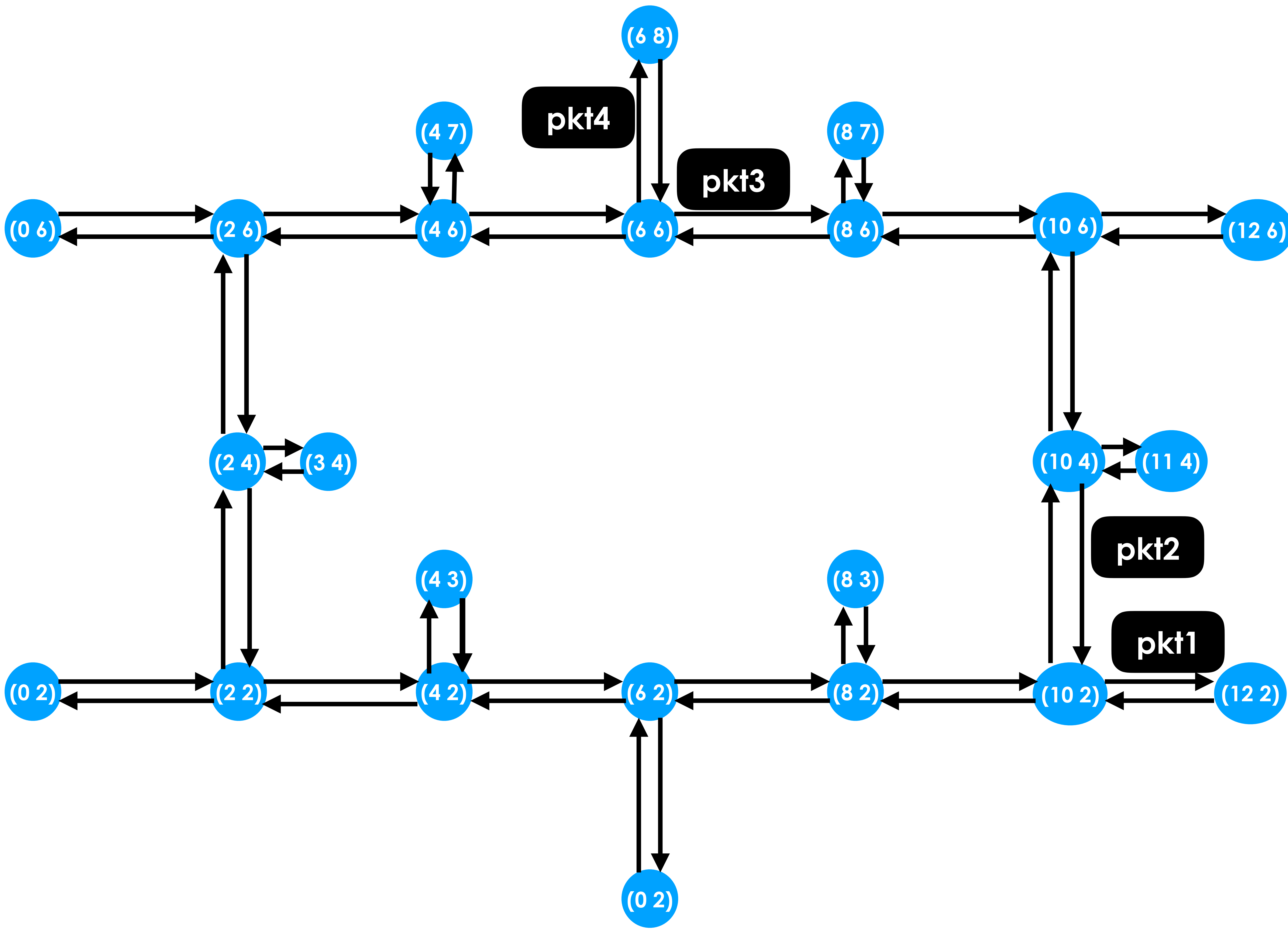
Packets leaving the network

- Maintain a global history for delivered packets
- Walk through the link table before each cycle of the system
- Check if packet present on the link has the final destination address as the current destination of link
- If so, update the global history and empty the link
- For example, if the current link table is

(((S12 (DONE (12 2) (0 2) DATA)) <= pkt1
(T9 (E (12 2) (6 8) DATA)) <= pkt2
(S26 (N (8 7) (3 4) DATA)) <= pkt3
(T20 (DONE (6 8) (11 4) DATA))) ...) <= pkt4

- Since some packets have arrived at the destination, they leave the network and are updated in the global history

Link table => ((S12 (DONE (12 2) (6 8) DATA)) <= pkt2
(T22 (DONE (8 7) (3 4) DATA))...) <= pkt3
global history => ((DONE (6 8) (11 4) DATA) <= pkt4
(DONE (12 2) (0 2) DATA)) <= pkt1



Packets leaving the network

- Maintain a global history for delivered packets
- Walk through the link table before each cycle of the system
- Check if packet present on the link has the final destination address as the current destination of link
- If so, update the global history and empty the link
- For example, if the current link table is

(((S12 (DONE (12 2) (0 2) DATA)) <= pkt1

(T9 (E (12 2) (6 8) DATA)) <= pkt2

(S26 (N (8 7) (3 4) DATA)) <= pkt3

(T20 (DONE (6 8) (11 4) DATA))) ...) <= pkt4

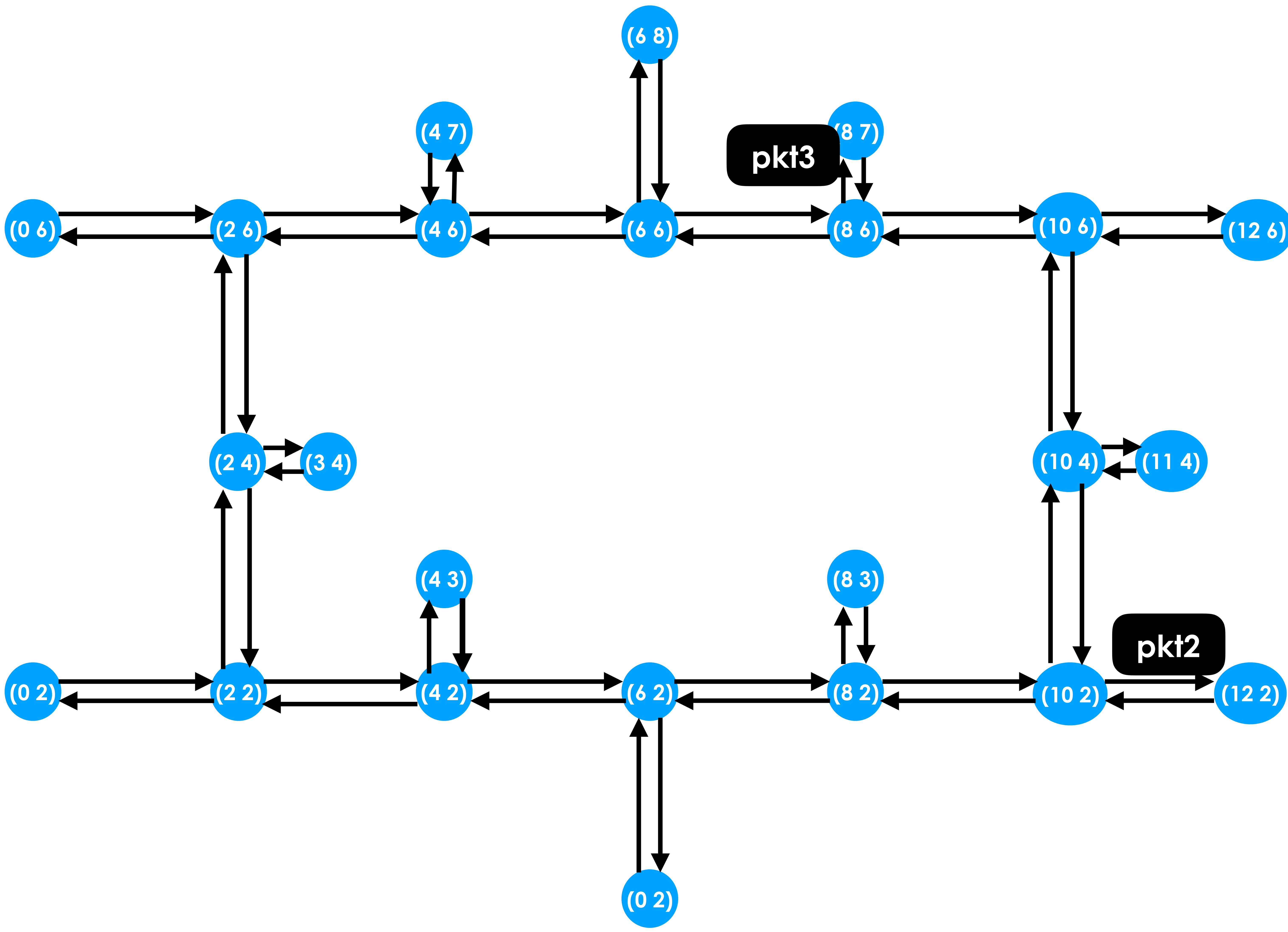
- Since some packets have arrived at the destination, they leave the network and are updated in the global history

Link table => ((S12 (DONE (12 2) (6 8) DATA)) <= pkt2

(T22 (DONE (8 7) (3 4) DATA))...) <= pkt3

global history => ((DONE (6 8) (11 4) DATA) <= pkt4

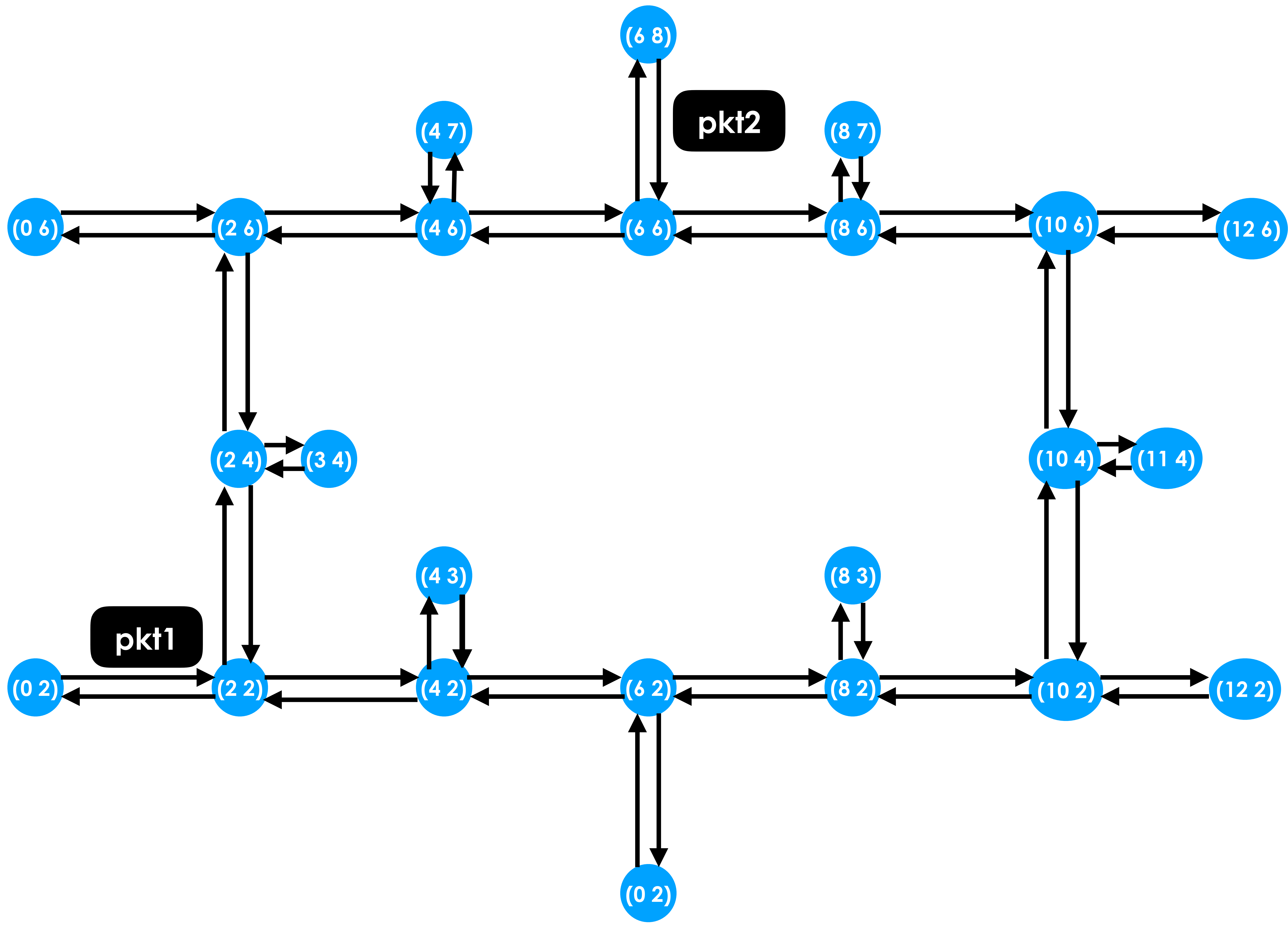
(DONE (12 2) (0 2) DATA)) <= pkt1



Example

- Link table before execution

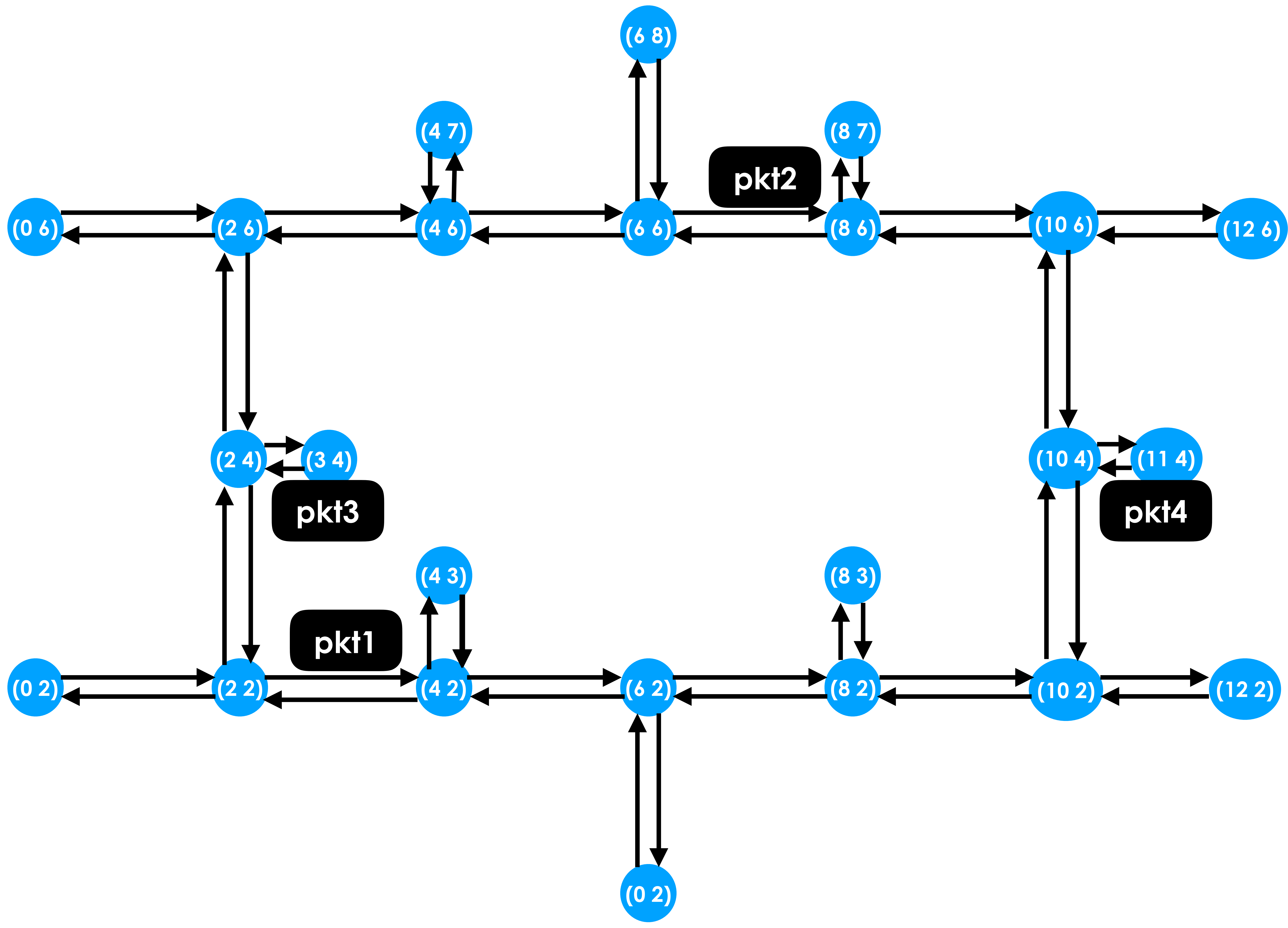
```
(((S2 (E (12 2) (0 2) DATA))     <= pkt1
  (T19 (E (12 2) (6 8) DATA))    <= pkt2
   ...))
```



Example

- Link table after 1 step and introduction of two packets

```
(((S4 (E (12 2) (0 2) DATA))      <=pkt1
   (S15 (N (8 7) (3 4) DATA))     <= pkt3
   (S17 (N (6 8) (11 4) DATA))    <= pkt4
   (S26 (E (12 2) (6 8) DATA))    <= pkt2
     ...))
```



Example

- Link table after 2 steps

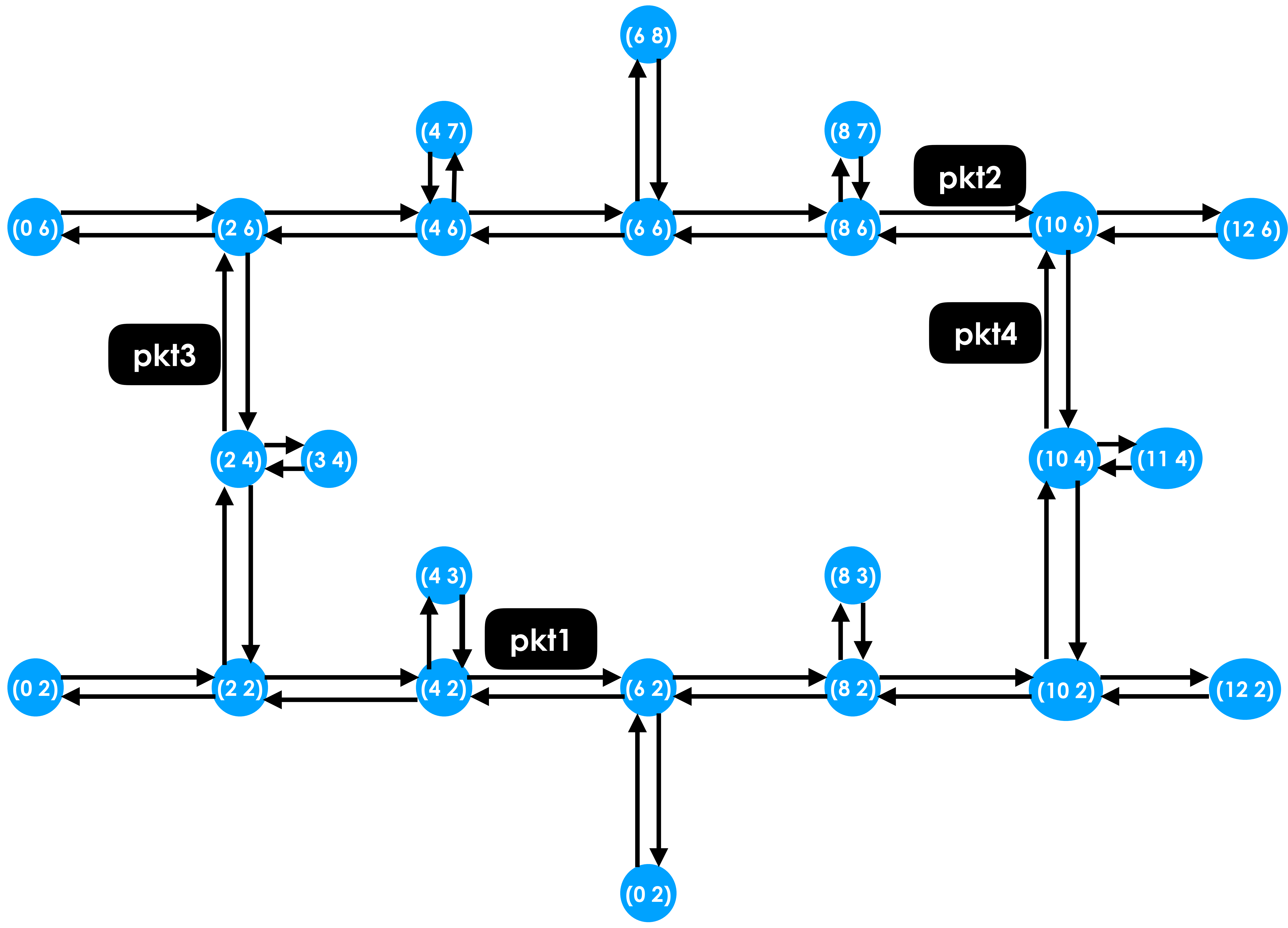
(((S6 (E (12 2) (0 2) DATA)) <=pkt1

(T14 (E (8 7) (3 4) DATA)) <= pkt3

(S28 (S (12 2) (6 8) DATA)) <= pkt2

(T16 (W (6 8) (11 4) DATA)) <= pkt4

...))



Example

- Link table after 3 steps

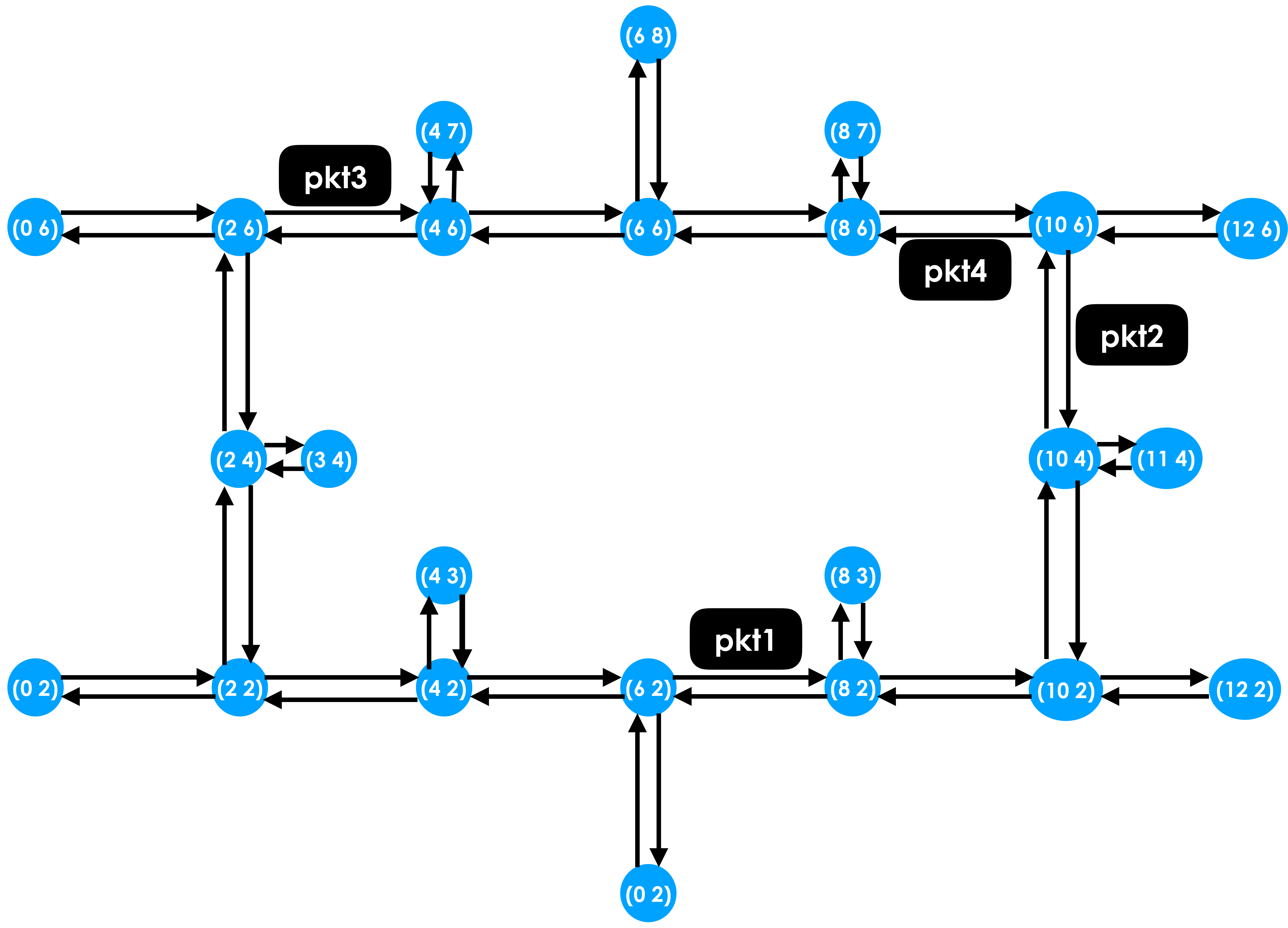
(((S8 (E (12 2) (0 2) DATA)) <=pkt1

(S22 (E (8 7) (3 4) DATA)) <= pkt3

(S27 (W (6 8) (11 4) DATA)) <= pkt4

(T15 (S (12 2) (6 8) DATA)) <= pkt2

...))



Example

- Link table after 4 steps

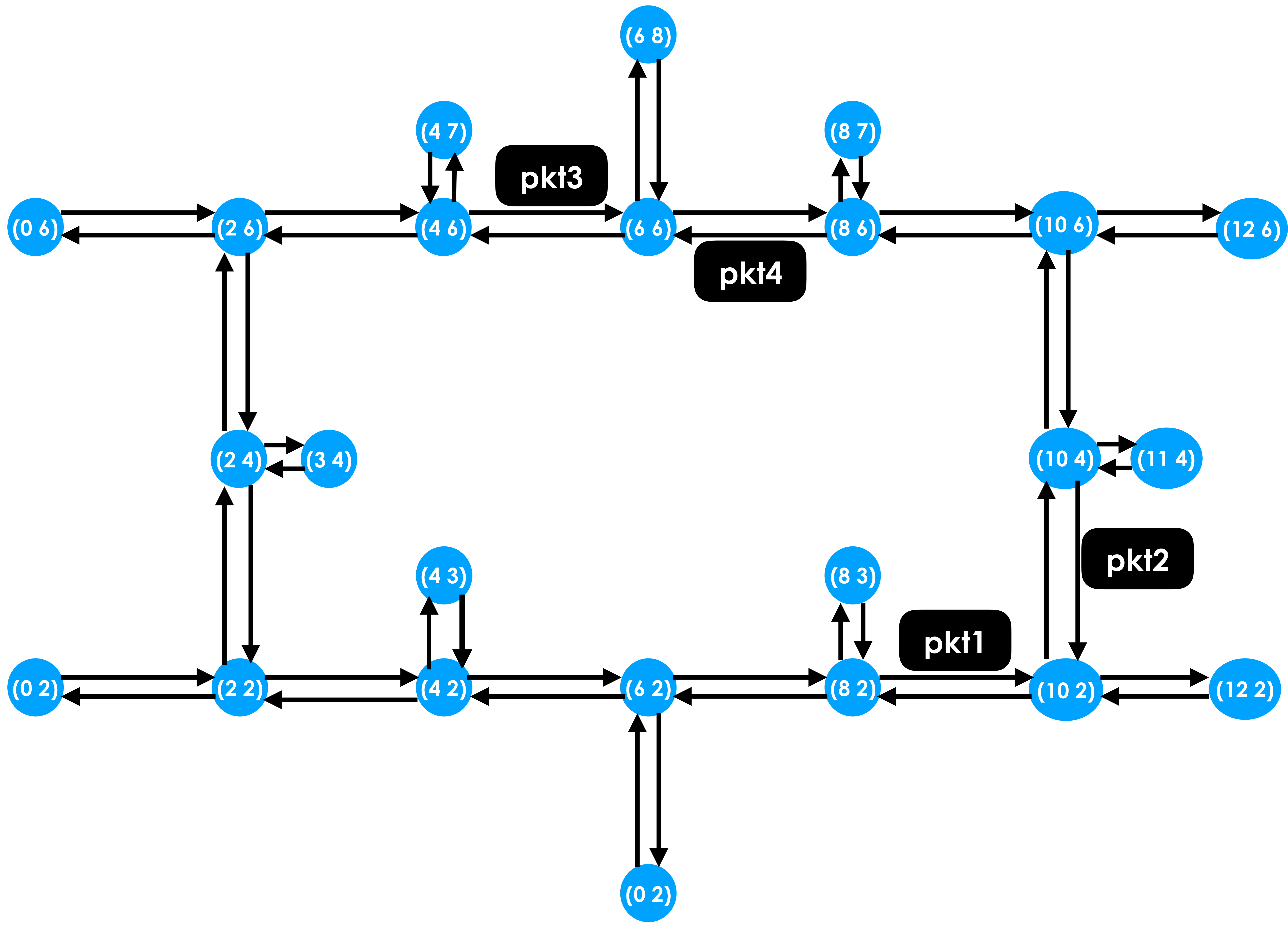
(((S10 (E (12 2) (0 2) DATA)) <=pkt1

(T9 (E (12 2) (6 8) DATA)) <= pkt2

(S24 (E (8 7) (3 4) DATA)) <= pkt3

(S25 (N (6 8) (11 4) DATA)) <= pkt4

...))



Example

- At this point, the arbiter picks between the competing packets, pkt1 and pkt2
- Link table after 5 steps and arbitration

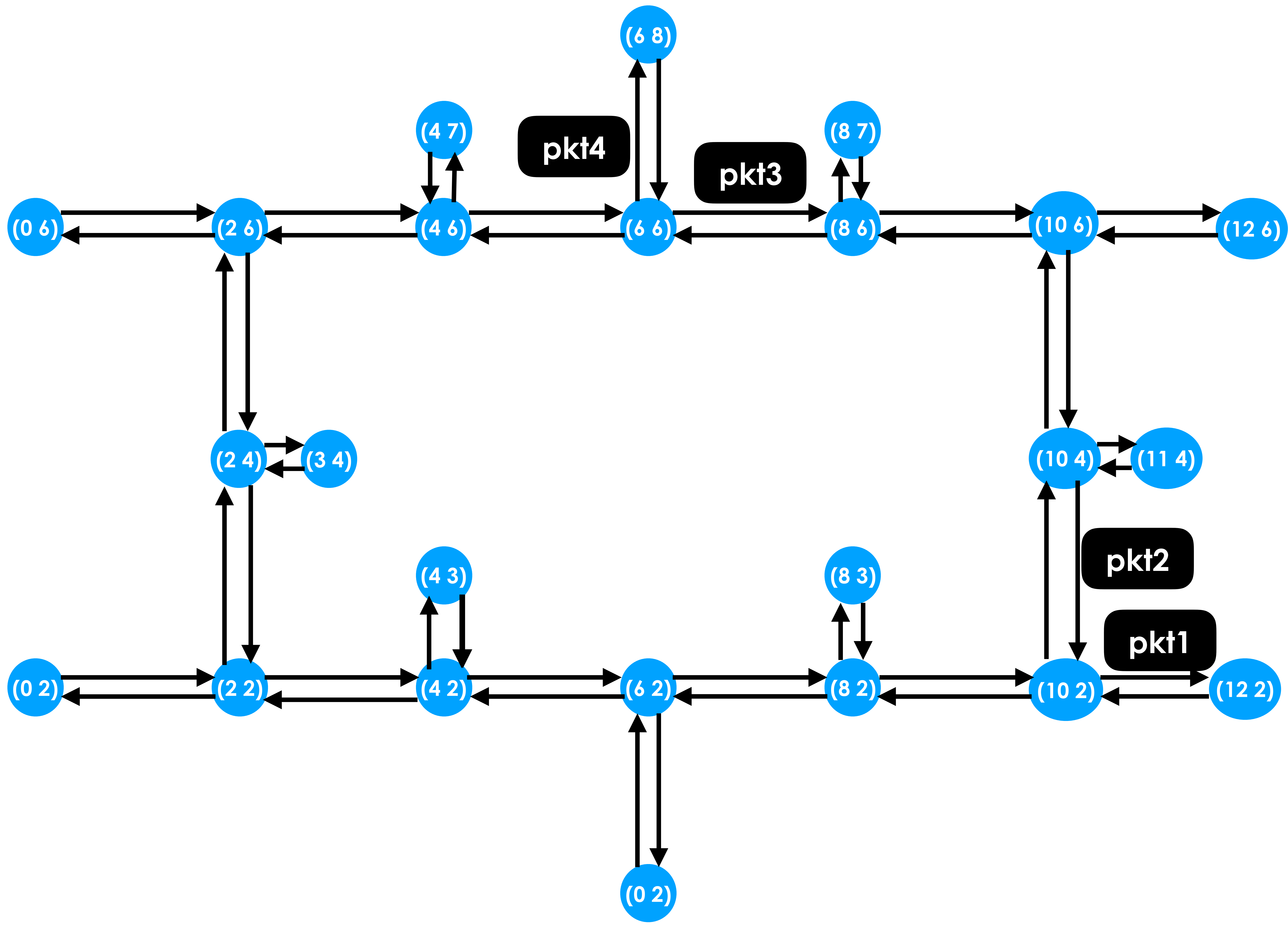
(((S12 (Done (12 2) (0 2) DATA)) <=pkt1

(T9 (E (12 2) (6 8) DATA)) <= pkt2

(S26 (N (8 7) (3 4) DATA)) <= pkt3

(T20 (Done (6 8) (11 4) DATA)) <= pkt4

…))



Example

- Since some packets have reached their final destination, they are added to global history
- Link table after 6 steps

(((S12 (Done (12 2) (6 8) DATA)) <= pkt2

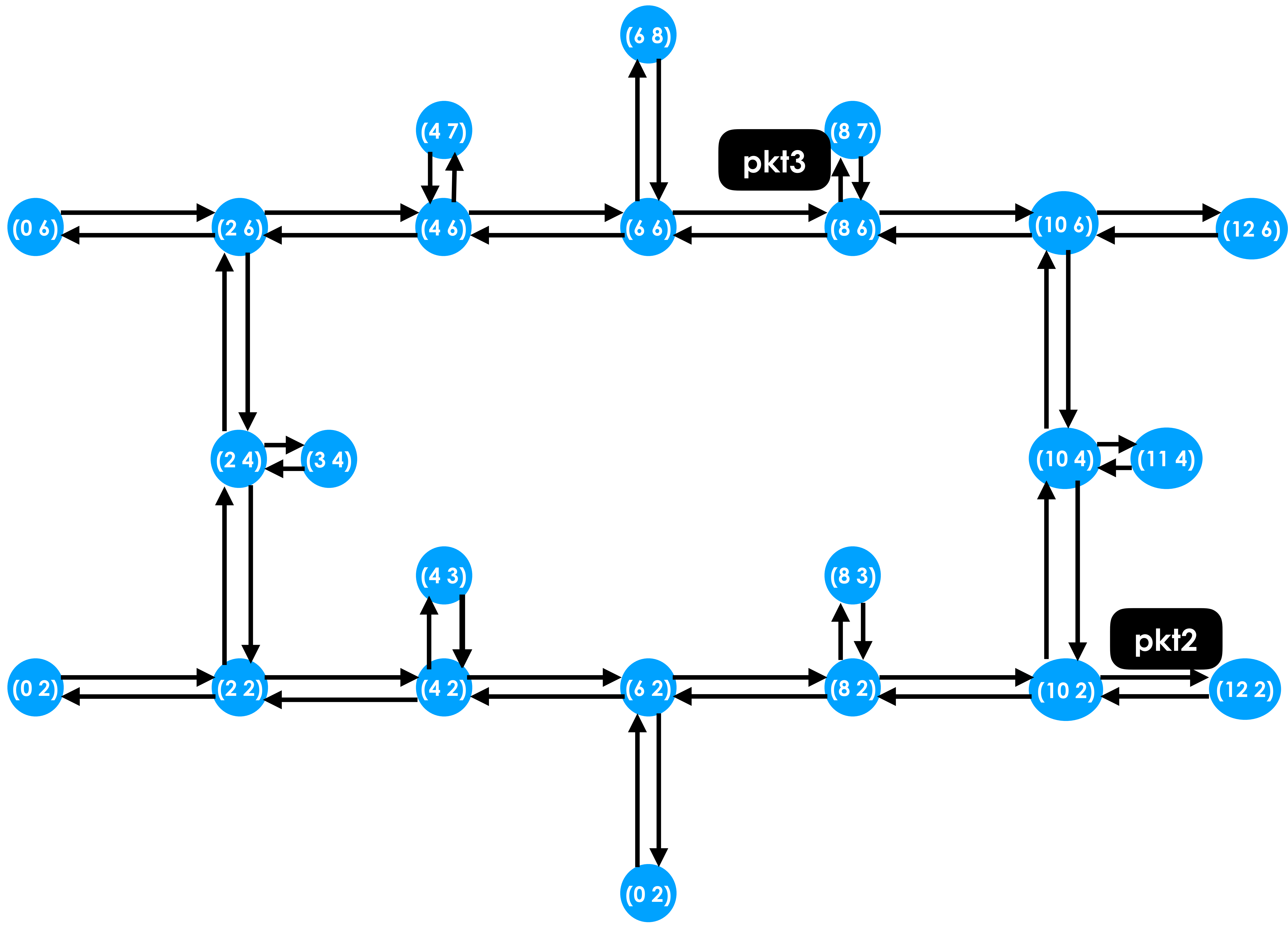
(T22 (Done (8 7) (3 4) DATA))   <= pkt3

…))

- Global history

(((6 8) (11 4) DATA)

(12 2) (0 2) DATA))))



Some Desired Properties

- Reachability
- Progress
- Packets are delivered

Reachability

- Every source can send a packet to any destination in the network
- By statically determining a path between any two junctions in the network, we can prove that packets can move between any two points in the network.
- Using the function, “find-path”, we can build the path that the packet will take between two junctions junctA and junctB.

```
(defthm reachability
   (implies  (and  (graphp g)
                   (junctp junctA g)
                   (junctp junctB g))
          (let ((p (find-path junctA junctB g)))
              (and (pathp p g)
                     (equal (car p)  junctA)
                      (equal (car (last p)) junctB)))))
```

Progress

- However, we can reason about progress better by using the reachability property

- By checking if a packet is making progress along its path, we can prove that there is packet movement

```
(defthm one-packet-progress
   (implies
         (and   (Single-step-guard input-list lt st g MS)
                  (packet-can-move pkt1 lt g))
         (<  (len (get-packet-path pkt1 (Single-step input-list lt st g MS)))
              (len (get-packet-path pkt1 lt))))
```

- The function, (packet-can-move pkt1 lt g)), means that the output link is empty and the packet is not waiting on an arbitration decision

Packets are delivered

Lemma: Without the introduction of new packets into the system and enough empty-inputs k, all the packets in the link table will reach its destination.

A possibility of such K could be,

(len K) = (product of all packet path-lengths

in the link table)

```
(defun-sk exists-k (output-stream lt inverse-map st g MS)
 (exists k
        (and (valid-input-stream k)
             (< 0 (len k))
             (destination-property
                  (car (multi-step k output-stream lt
                                     inverse-map st g MS)))
                  g))))
(defthm packet-destination-thm
 (implies  (and  (graphp g)
                 (linktable-validp lt g)
                 (statemap-validp st g)
                 (inverse-map-p inverse-map g)
                 (valid-output-stream output-stream g)
                 (bitp MS))
        (exists-k output-stream lt inverse-map st g))
 :hints (("Goal" :use ((:instance exists-k-suff)))))
```

Future Work

- Work on proving all the above properties
- Continue to enhance model to depict other functionalities of HexNet

```
(defun arbiter (outlink inputs st g)
  "Makes decision about ties, by accessing the decision made at previous steps."
   (if (endp inputs)
             (mv nil st)
        (if (endp (cdr inputs))
               (mv (car inputs) st)
           (let* ((entry (assoc-equal outlink st))
                    (pref (cdr entry)))
             (if (and pref (member-equal pref inputs))
                   (let* ((new-pref (car (remove1-equal pref inputs)))
                          (new-st (update-alist outlink new-pref st)))
                    (mv pref new-st))
               (let* ((new-pref (cadr inputs))
                    (new-st (update-alist outlink new-pref st)))
                 (mv (car inputs) new-st)))))))
```

```
(defun Routing (past curr final g)
  "Calculates direction, one step in advance."
   (let ((dests (remove-equal past (sources curr g))))
    (cond
          ((atom dests)    (if (equal curr final)                                   ;; Case 1: If there is no junction in the list Dests, then the packet is at its destination or there is an error
                                 'Done
                             (cw "Bad route")))
        ((atom (cdr dests))    (get-direction curr (car dests) g))                  ;; Case 2: If there is 1 junction in the list Dests, then return the direction to the junction
        ((atom (cddr dests))                                                        ;; Case 3: If there are 2 junctions, determine which will take the packet  closest to the final destination
address
                              (let*    ((dest1 (car dests))    (dest2 (cadr dests))
                                          (dest1-x (car dest1))    (dest1-y (cadr dest1))
                                          (dest2-x (car dest2))    (dest2-y (cadr dest2))
                                         (dir1 (get-direction curr dest1 g))
                                         (dir2 (get-direction curr dest2 g))
                                        (curr-x (car curr))   (curr-y (cadr curr))
                                        (final-x (car final))    (final-y (cadr final)))
                               (if (isTerminal dest1 g)
                                      (if (equal dest1 final)
                                             dir1
                                           (if (isTerminal dest2 g)
                                                 (if (equal dest2 final)        dir2        (cw "Bad route"))
                                            dir2))
                                 (if (isTerminal dest2 g)
                                         (if (equal dest2 final)    dir2   dir1)
                                     (cond      ((< curr-y final-y)        (if (= dest1-y dest2-y)
                                                                                   (if (< (abs (- dest1-x final-x))  (abs (- dest2-x final-x)))          dir1        dir2)
                                                                               (if (< dest1-y dest2-y)     dir2     dir1)))
                                                ((> curr-y final-y)       (if (= dest1-y dest2-y)
                                                                                   (if (< (abs (- dest1-x final-x))  (abs (- dest2-x final-x)))          dir1        dir2)
                                                                               (if (< dest1-y dest2-y)      dir1     dir2)))
                                                ((< curr-x final-x)       (if (< dest1-x dest2-x)        dir2        dir1))
                                                ((> curr-x final-x)       (if (< dest1-x dest2-x)        dir1         dir2))))))))))
```

```
(defun update-link (junct dest lt st g MS)
 "Moves the packet on an input link to an output link"
      (if MS                ;; MS is the model state. This indicates if there is an error in the current state
           (mv MS lt st)
         (let*   ((outlink  (get-output-link junct dest g)))
           (if (get-packet outlink lt g)                              ;; The receiving link is full
                 (mv MS lt st)
              (let*    ((stack (remove-equal dest (sources junct g)))
                      (inputs (current-inputs junct dest stack lt g)))           ;; Check any packets on the input (sending) links of the junction
                 (if  (endp inputs)
                      (mv MS lt st)
                   (let*   ((result (mv-list 2 (arbiter outlink inputs st g)))         ;;  Run inputs through the arbiter to choose one and store decision
                            (input (car result))
                            (new-st (cadr result)))
                     (if (get-packet input lt g)                              ;;  Check that the packet on the link is valid
                         (let*   ((pkt (get-packet input lt g))
                                  (final (cadr pkt))
                                  (turn_signal (routing junct dest final g)))      ;; Run routing function to get the next turn signal
                           (if  (turn_signalp turn_signal)
                               (let*  ((new-pkt  (list turn_signal final caddr pkt))                    ;; repack packet with new turn_signal
                                      (new-link-state (update-alist outlink (list new-pkt) lt))          ;; update the receiving link with new packet
                                      (new-lt (update-alist input nil new-link-state)))                ;;  remove old packet from the sending link
                                 (mv MS new-lt new-st))
                             (mv “Bad Route” lt new-st)))      ;; Error message from the routing function, It can not return a valid turn signal
                        (mv "Invalid packet" lt st)))))))))        ;; Error message while reading the packet, the packet is not well-formed
```

```
(defun process-junct (junct neighbors lt st g MS)
 "Updates the links with packets connected to a junction"
  (declare (xargs :guard (and (graphp g)
                              (linktable-validp lt g)
                              (junctp junct g)
                              (statemap-validp st g)
                             (true-listp neighbors)
                              (subsetp-equal neighbors (sources junct g))))
   (if MS
       (mv MS lt st)
     (if (endp neighbors)
         (mv MS lt st)
       (let* ((dest (car neighbors))
              (result (mv-list 3 (update-link junct dest lt st g MS)))
              (new-MS (car result))
              (new-lt (cadr result))
              (new-st (caddr result)))
        (process-junct junct (cdr neighbors) new-lt new-st g new-MS)))))
```

```
(defun Single-step (junct-list lt st g MS)
 "Run the interpreter once"
 (declare (xargs :guard (and (graphp g)
                             (linktable-validp lt g)
                             (statemap-validp st g)
                             (true-listp junct-list)
                             (subsetp-equal junct-list (all-juncts g))))
 (if MS
     (mv MS lt st)
   (if (endp junct-list)
       (mv MS lt st)
     (let* ((junct (car junct-list))
            (neighbors (sources junct g))
            (result (mv-list 3 (process-junct junct neighbors lt st g MS)))
            (new-MS (car result))
            (new-lt (cadr result))
            (new-st (caddr result)))
       (Single-step (cdr junct-list) new-lt new-st g new-MS)))))
```

```
(defun add-new-packets (input lt g)
 (declare (xargs :guard (and (graphp g)
                             (linktable-validp lt g)
                             (valid-input input g))))
 (if (endp input)
     lt
   (let* ((entry (car input))
          (final (car entry))
          (source (cadr entry))
          (data (caddr entry)))
     (if (isTerminal source g)
         (let* ((junctB (car (sources source g)))
                (turn_signal (routing source junctB final g)))
           (if (turn_signalp turn_signal)
               (let* ((pkt (list turn_signal final source data))
                      (outlink (get-output-link source junctB g))
                      (new-lt (update-alist outlink (list pkt) lt)))
                 (add-new-packets (cdr input) new-lt g))
             (add-new-packets (cdr input) lt g)))
       lt))))
```

```
(defun remove-finished-packets (output-stream lnk-lst lt g)
 (declare (xargs :guard (and (graphp g)
                            (linktable-validp lt g)
                            (valid-output-stream output-stream g))))
 (if (atom lnk-lst)
     (mv lt output-stream)
     (if (get-packet (car lnk-lst) lt g)
         (let* ((pkt (get-packet (car lnk-lst) lt g))
                (turn_signal (car pkt)))
           (if (equal turn_signal 'Done)
               (let ((new-output-stream (cons (get-packet (car lnk-lst) lt g)
                                              output-stream))
                     (new-lt (update-alist (car lnk-lst) nil lt)))
                 (remove-finished-packets new-output-stream (cdr lnk-lst) new-lt g))
             (remove-finished-packets output-stream (cdr lnk-lst) lt g)))
       (remove-finished-packets output-stream (cdr lnk-lst) lt g))))
```

```
(defun multi-step (input-stream output-stream lt inverse-map st g MS)
 "Run the system multiple times"
 (declare (xargs :guard (and (graphp g)
                             (linktable-validp lt g)
                             (statemap-validp st g)
                             (inverse-map-p inverse-map g)
                             (valid-input-stream input-stream g)
                             (valid-output-stream output-stream g)))
 (if MS
     (mv MS nil)
   (if (endp input-stream)
       (mv lt output-stream)
     (let* ((result1 (mv-list 2 (remove-finished-packets output-stream (strip-cars lt) lt g)))
            (trans1 (car result1))
            (new-output-stream (cadr result1))
            (junct-list (get-junct-list trans1 inverse-map g))
            (result (mv-list 3 (Single-step junct-list trans1 st g MS)))
            (new-MS (car result))
            (trans2 (cadr result))
            (new-linktable (add-new-packets (car input-stream) trans2 g))
            (new-statemap (caddr result)))
       (multi-step (cdr input-stream)
                   new-output-stream
                   new-linktable
                   inverse-map
                   new-statemap
                   g
                   new-MS)))))
```