Document Number: IKN-2002-1

Chapter 9, Fairness

March 4, 2002

Harmen R. van As, Arben Lila, Guenter Remsak, Jon Schuringa
Vienna University of Technology, Austria

Email:
harmen.r.van-as@tuwien.ac.at
arben.lila@tuwien.ac.at
guenter.remsak@tuwien.ac.at
jon.schuringa@tuwien.ac.at

Abstract: This text describes the fairness algorithm in a combined greedy and cyclic reservation MAC protocol for ring networks. It performs at the theoretical fair limits and therefore exhibits excellent performance in terms of throughput, end-to-end delay, guarantees of service level agreement, and traffic dynamics. Other major features are: the support of multiple service classes, the support of heterogeneous link rates, no measurements on the links, no buffer thresholds, self-adaptive.

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art.

Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason IEEE and the members of its technical committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331

IEEE Standards documents are adopted by the Institute of Electrical and Electronics Engineers without regard to whether their adoption may involve patents on articles, materials, or processes. Such adoption does not assume any liability to any patent owner, nor does it assume any obligation whatever to parties adopting the standards documents.

Status summary

Contacts

Harmen R. van As, Arben Lila, Guenter Remsak, Jon Schuringa
Vienna University of Technology, Austria
Email:
harmen.r.van-as@tuwien.ac.at
arben.lila@tuwien.ac.at
guenter.remsak@tuwien.ac.at
jon.schuringa@tuwien.ac.at

Table of contents

List of figures

List of tables

Change history

The following table shows the change history for this user's manual.

Version 1.0 (date)

Original version.

Version X.X (date)

Category	Description
Editorial	Description here
Technical	Description here

9. Fairness

9.1 Introduction

Fairness control mechanisms for rings can be classified in global and link fairness mechanisms. Traditional medium access control protocols are based on global fairness, where each station obtains the same throughput, independently whether a node disturbs flows of other nodes or not. Today, advances in microelectronics allow the design of more sophisticated link or bottleneck fairness mechanisms, potentially resulting in much high network throughputs.

Definition of Global fairness: Fairness based on a mechanism that allows nodes to share the same amount of the transmission capacity of the ring, independently whether their traffic interfere or not.

Definition of Link fairness: Fairness based on a mechanism that coordinates ring access of only those nodes that interact during their packet transfer. Thus, all nodes that do not interfere are not throttled in their performance.

In Figure 9.1, it can be seen that in the case of global fairness the flow from station 5 to station 6 is throttled down to a rate of 0.5 because of the bottleneck on the link between stations 1 an 2. In case of link fairness, this unnecessary throttling does not take place.

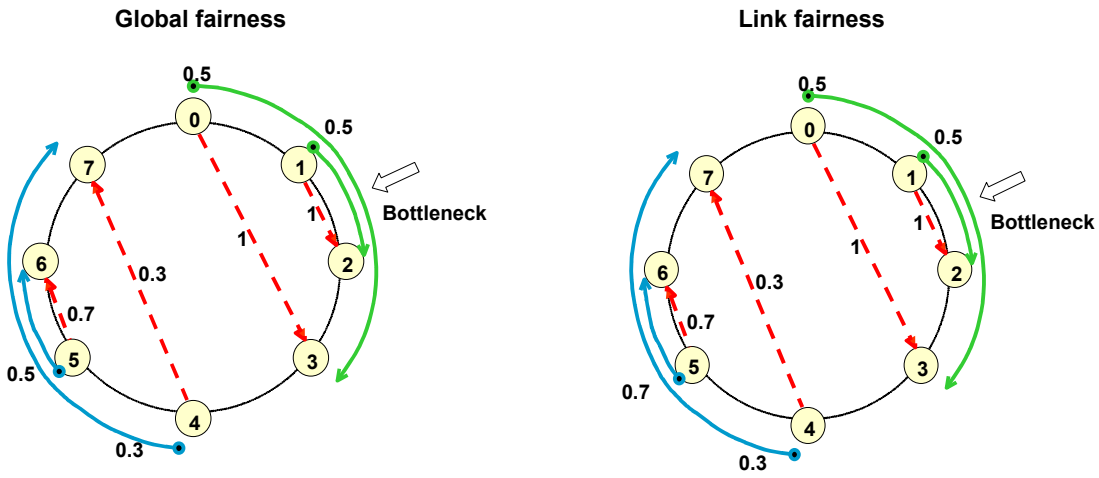
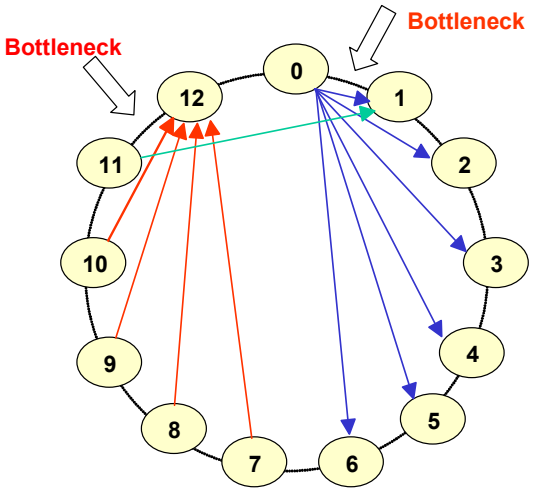


Figure 9.1 Global and link fairness on a single ringlet

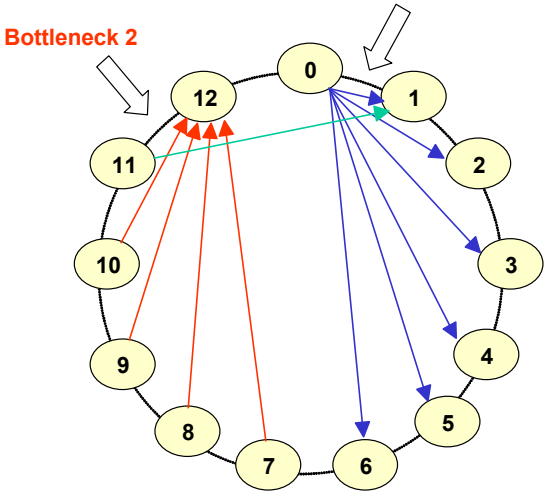
Link fairness can be achieved in multiple ways, depending on the definition of link fairness. One can base link fairness on the demands of competing flows, where all are throttled proportionally (Figure 9.2), or it can be based on the number of flows over a bottleneck (Figure 9.3).



Fairness Definition 1

Source	Sink	Rate	Fair
0	1	0.01	0.01
0	2	0.01	0.01
0	3	0.01	0.01
0	4	0.01	0.01
0	5	0.01	0.01
0	6	0.01	0.01
7	12	0.1	0.071
8	12	0.1	0.071
9	12	0.1	0.071
10	12	0.1	0.071
11	1	1	0.71

Figure 9.2 Fair station throughputs in case of fairness definition 1 (proportional throttling)



Fairness Definition 2

Source	Sink	Rate	Fair
0	1	0.01	0.01
0	2	0.01	0.01
0	3	0.01	0.01
0	4	0.01	0.01
0	5	0.01	0.01
0	6	0.01	0.01
7	12	0.1	0.1
8	12	0.1	0.1
9	12	0.1	0.1
10	12	0.1	0.1
11	1	1	0.6

Figure 9.3 Fair station throughputs in case of fairness definition 2 (throttling related to the number of flows)

Furthermore, one distinguishes between reactive and proactive control mechanisms. In reactive control, a node detecting congestion on its outgoing link typically sends a backpressure control packet in the opposite direction to its upstream nodes enforcing them to stop transmission or enforcing to reduce their rate. In proactive fairness control, a control packet circulates around the ringlet to coordinate the individual source-destination flows of each node. For the content of the control packet several variations are possible. One possibility is that each station i (i = 1,…, N) measures the number of bytes of each flow f_{ij} from source i to

destination j on its outgoing link i (i = 1, …, N) during the cycle time T_c of the control packet. When the control packet arrives at station i, it calculates the fair rate r_i over its outgoing link i and writes the result into the data field of link i in the control packet. Since each station does this measurement, all stations are cyclically updated with all the current fair link rates r_i on the ring. For a dual ring, there is on control packet on each ringlet. Control packets can either circulate in the same direction of the data flow or in the opposite direction. In the latter case, one ringlet is used for the data flow and the other ringlet for its control.

In this proposal, however, control and data packets flow in the same direction. This has the advantage, that in case of multiple parallel ringlets, there is a clear and simple association between data and control packets belonging to a ringlet. In addition, we use no measurement data but in stead the current traffic load waiting in each station to be transmitted. Due to this, the proactive control is based on the latest flow information, thus allowing to dynamically adapt in the fastest way to traffic changes. The traffic pattern may even completely change in every cycle and the mechanism is still able to react properly.

9.2 Fairness Algorithm

The fairness algorithm described in this section computes the fair rates for each source destination flow from a cyclically updated demand matrix. It assigns the fair rates in a theoretical optimal way, i.e., all flows get their maximum possible rate. Other features of the algorithm are:

Multiple different link capacities:
The algorithm assigns all rates in such a way that bottlenecks do not occur, even in the case where multiple link capacities exists on one ringlet.

Multiple traffic classes:
The algorithm assigns fair rates for high *and* low traffic. It uses a mechanism to control the total amount of high and low priority on each link, this prevents high priority taking all available capacity when large amounts of high and low traffic are simultaneously being scheduled. Additionally, the algorithm is aware of provisioned bandwidth connections.

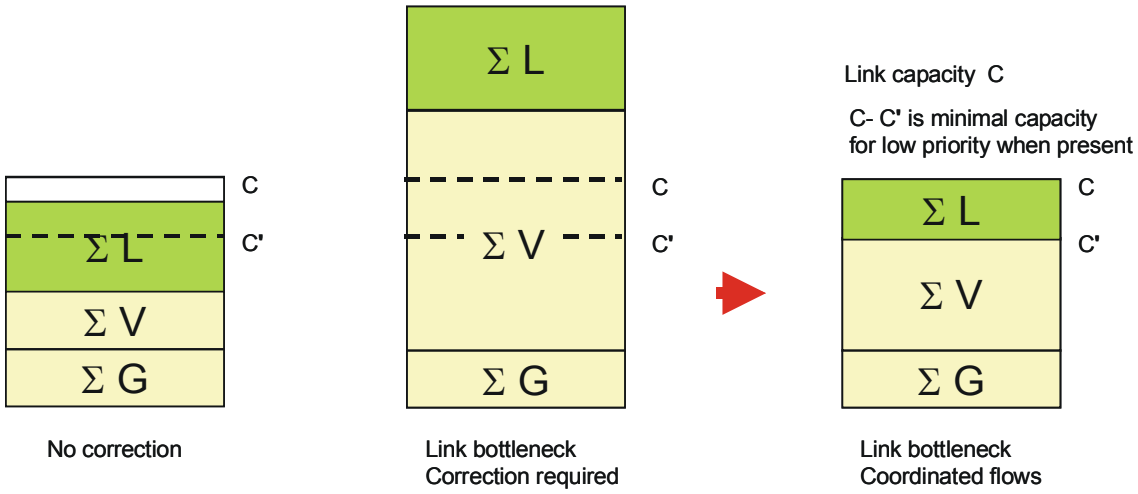


Figure 9.4 Fairness of multiple traffic classes

The fairness algorithm is executed at each node at each calculation interval. The input to the algorithm is the local flow table that is continuously being updated by a single control packet. How this is done will be explained in Section 9.3 “Control Packet”.

The fair rates are calculated in three steps:

- Step 1

In the first step, high priority is being assigned up to the available link capacity multiplied by a factor *f_highbound*. This factor controls the minimum amount of low priority traffic on each link. If, for example, *f_highbound* equals 0.9, then high priority can maximally take up to 90% of the available capacity if there is more than 10% low priority. If there is only high priority traffic, then this traffic can of course take 100% of the link capacity, the same is true for low priority.
Available link capacity is defined by the total capacity minus the amount of capacity taken by the sum of all fixed connections over the link.

- Step 2

The second step is to assign low priority up to all available capacity. Assume that both high and low have up to 100% of the link capacity to send and *f_highbound* equals 0.9 (as before), then low priority will get its 10% in this step.

- Step 3

The last step is to assign the remaining high priority traffic up to full capacity.

Top level Pseudo Code:

```
MakeFair_1(){

   // init
   table->copyFrom(tableOriginal)
   allowed->toZero()

   //step 1:
   //Assign High Priority until highbound
   forall Links i {
      availableCap = linkCap[i] - fixedCap[i]
      fairData[i].remainingCapacity = availableCap * f_highBound
   }
   makeFair_2(HighPrio)

   // step 2:
   //  Assign low until total available capacity
   forall Links i{
      availableCap = linkCap[i] - fixedCap[i]
      fairData[i].remainingCapacity += availableCap * (1.0-f_highBound)
   }
   makeFair_2(LowPrio)

   // step 3:
   // Assign remaining high
   // The problem here is that our demand table is overwritten in step 1
   // So we copy the table from the original table and substract what
   // we allowed in step 1.
   table->copyFrom(tableOrig)
   table->subtract(allowed)
   makeFair_2(HighPrio)
}
```

fairData is an array that holds the following information for each link:

- nDemand: An integer denoting the number of source destination demands (flows) over the link.

- flow: At initialization, this variable is set to the total traffic demand over the link (sum of all flows over the link). During algorithm execution, this value is decreased.
- remainingCapacity: Initialized to the available link capacity,

We continue with the function "Makefair_2", the core of the fairness algorithm. The idea is to assign at each step the smallest possible amount of bytes to the flows, do this as long as there are bottlenecks. The code below uses the array "flows", which contains all source-destination pairs over the bottleneck link. "Table" holds the input data; "allowed" holds the output when the algorithm finishes.

```
MakeFair_2(int prio){
   init()
   Do{
      bottleneck = highestBottleneckLink()
      if (bottleneck >=0 ) {
         somethingDone  = false
         fairRate       = calcFairRate(bottleneck)

         // first assign those flows that want to send
         // less than the calculated fair rate
         forall flows i over bottleneck {
            demand = table->get(  flows[i].from, flows[i].To , prio)
            if ( demand < fairRate) {
               allowed->plus( flows[i].from, flows[i].To , prio, fairRate)
               table     ->set(  flows[i].from, flows[i].To , prio, 0)
               updateFairnessBetween(pair, fairRate, fairRate)
               somethingDone = true
            }
         }

         // if we assigned flows in the previous code section
         // (i.e. when somethingDone is true) we are done (at
         // least for now with this bottleneck)
         // otherwise assign the fair rates to all flows over
         // the bottleneck
         if (!somethingDone){
            forall flows i over bottleneck {
               value = tbl->get(flows[i].from, flows[i].to, prio)
               if (value>0) {
                  allowed->plus( flows[i].from, flows[i].to,
                                 prio, fairRate)
                  table     ->set(  flows[i].from, flows[i].to, prio, 0)
                  updateFairnessBetween(flows[i].from, flows[i].to ,
                                        fairRate,value)
               }
            }
         }
      }
   } while  (bottleneck>0)

   // copy remaining demands (since they are not involved in a
   // bottleneck
   for (i=0;i<nrNodes;i++)     for (j=0;j<nrNodes;j++)
        allowed->plus(i,j,prio,table->get(i,j,prio));
}
```

Init calculates the flow and the number of demands on each link.

```
void init(){
   forall links i
      fair[i].flow     = 0
      fair[i].nDemand  = 0
      forall flows j over i {
           value = tbl->get(flows[i].from, flows[i].to, prio)
           if (value>0) {
              fair[i].flow      += value
              fair[i].nDemand ++
           }
      }
}
```

The function "highestBottleneckLink" returns the link id of the link that is the strongest bottleneck, i.e., from all links where the demand is higher that available capacity; it returns the link id with the highest number of flows passing over it. The function returns –1 when there are no bottlenecks on the ringlet.

„calcFairRate" is a straightforward function:

```
double calcFairRate(i){
   return fairData[i].remainingCapacity / fairData[i].nDemands;
}
```

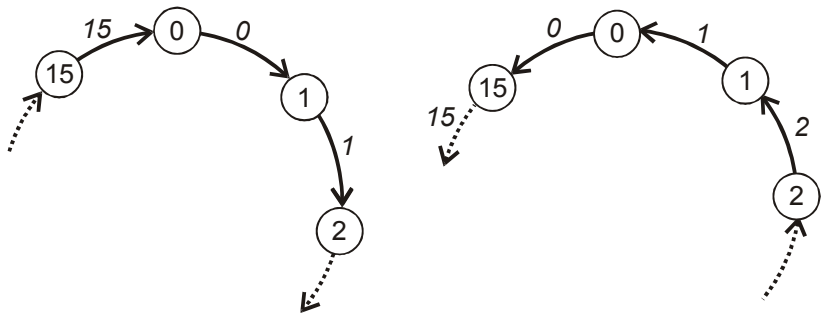


Figure 9.5 Link id's on both ringlets

The function "updateFairnessBetween" updates the fairness variables on all links between fromNode and toNode. The id of a link is the same as the node id where the link originates. This is true for both ringlets (see Figure 9.5). The following functions loops through all link id's between toNode and fromNode. Note that we have an amount1 and an amount2, the first one is the amount to decrease the remainingCapacity, the second to decrease the flow. These two amounts can be equal in case a flow has a smaller demand than the calculated fair rate.

```
void updateFairnessBetween(fromNode, toNode, amount1, amount2){
   int start,end;
   if (RI==1){
      // anti clockwise, we use a „trick" here
      start = (toNode   + 1)%nrNodes;
      end   = (fromNode + 1)%nrNodes;
   } else {
      // clockwise
      start     = fromNode;
```

```
      end     = toNode;
   }

   while (start!=end){
      fair[start]->remainingCapacity -= amount1;
      fair[start]->flow              -= amount2;
      fair[start]->nDemand --;
      start = (start+1)%nrNodes;
   }
}
```

Example:

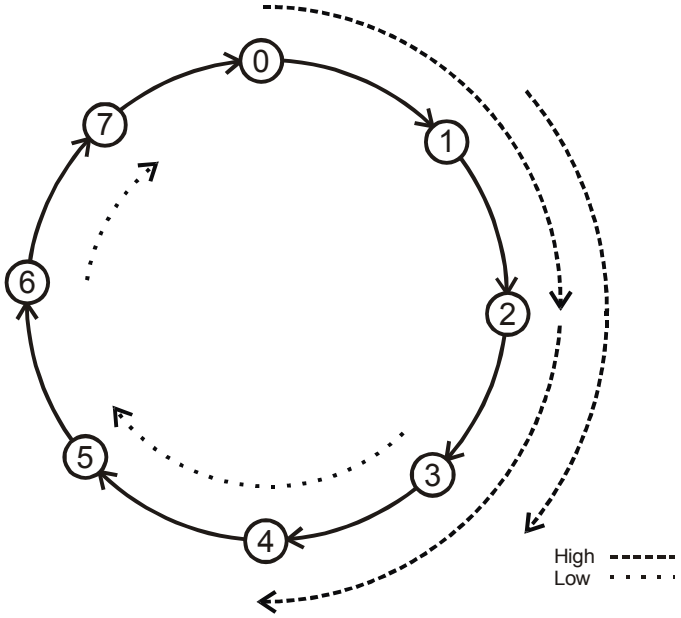


Figure 9.6 All flows want to send at link capacity

In Figure 9.6 we see a single ringlet with 5 flows. Assume our *f_highbound* equals 0.9, then the fairness algorithm assigns in the first step all high priority flows up to 90% of the available capacity. This can be seen in Table 9.1, where all high priority flows get 45% because they all have to share a link on their path with one other high priority flow.
The second step adds the low priority traffic; it leaves the assigned high priority rates unchanged. The last step gives the high priority flows, where possible, the remaining capacity.

Flow	Step 1	Step 2	Step 3
0→2	45%	45%	50%
1→3	45%	45%	50%
2→4	45%	45%	45%
3→5		55%	55%
6→7		100%	100%

Table 9.1 Rate Assignments at each step

9.3 Control Packet

9.3.1 Introduction

The control packet that is circulating on the ringlet contains information that is used by the fairness algorithm described in Section 9.2. Each node maintains a table with the amount of bytes for each source-destination pair; each source node advertises these values. The control packet distributes this information, however, the amount of information in a network with 256 nodes leads to a very big control packet. Since one big packet is impractical, we cut the control packet in smaller pieces. This, of course, does not reduce the amount of data being transmitted, but has advantages from both theoretical and practical viewpoints. Recall that the fairness algorithm is based on the fact that all nodes have the same information, so the smaller the control packet that is circulating and updating the information in the nodes, the better are they synchronized at any time.
As a result, the control packet is continuously circulating in the same direction as the data flow, and the packet is holding only a part of the complete table. How this is done will be described in the following sections.

9.3.2 Control Packet Format

Before defining the control packet format, we must define how the complete table with all flows is organized. As said in the previous section, the control packet is holding only a certain part of the complete table. So each time when a control packet arrives, the node must know where to place the new information in its table.
For a network with N nodes and two counter propagating ringlets, we define integer M to be N/2. The reason for this is that we assume shortest path routing (based on the number of hops) and allow, in case of equal costs, both possibilities. For a network with only one ringlet, M is defined to be N-1.
The table on the clockwise ring is defined as follows, where each entry contains the traffic demand for both high and low priority traffic.

0 to 1	0 to 2	0 to...	0 to M
1 to 2	1 to 3	1 to …	
…	…	…	…
N-1 to 0	N-1 to 1	N-1 to …	N-1 to M-1

Table 9.2 Clockwise Ringlet Table

And the table for the counter propagating ringlet:

0 to N-1	0 to N- 2	0 to...	0 to N- M
N-1 to N-2	1 to N- 3	1 to …	
…	…	…	…
1 to 0	1 to N-1	N-1 to …	N-1 to M-1

Table 9.3 Counter Clockwise Ringlet Table

Example:

On a ring with 12 Nodes, the table used at the clockwise ringlet (RI=0) is organized as follows. N and M are 12 and 6 respectively. In Figure 9.7, the first part of the table is shown. The number in the upper right corner of each square is the index; the other numbers denote the source-destination flow. The general functions that operate on the tables will be given in the next sections, for now we note that given a certain index *i*, the source node id is given by *i*/M, the destination node id by ((*i* mod M)+sourceID+1) mod N.

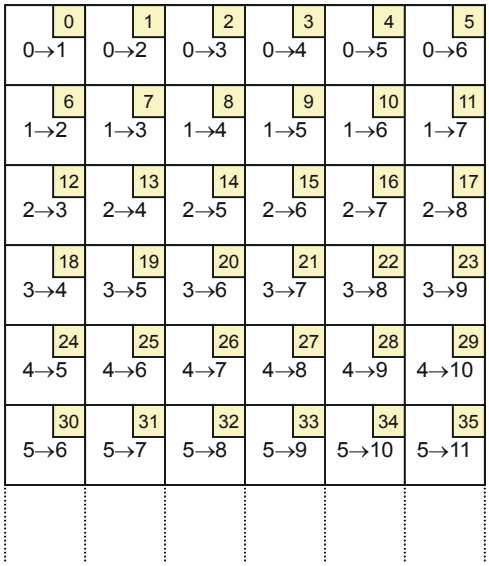


Figure 9.7 Source-Destination Pairs

Now we come to the specific control packet fields:

Field	Type	Explanation
Offset	int (2 bytes)	The offset of the "data" field bytes in the complete table
ValidEntries	int (2 bytes)	The number of valid entries contained by the "data" field.
Data	*Byte	Array holding the actual data. The length is a configurable system parameter.

Table 9.4 Control Packet Fields

As an example, consider Figure 9.8. The blue squares denote the place of the control packet data in the table. In this case offset equals 14 and validEntries equals 8. The first data entry (data [0]) in the packet contains the flow information from node 2→5 (this can be seen in Figure 9.7).

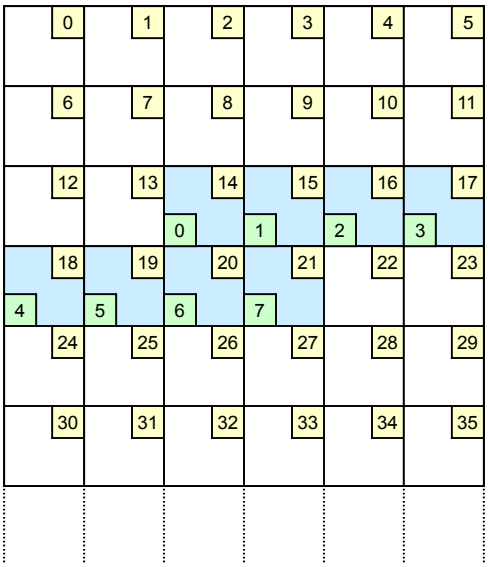


Figure 9.8 Position of the packet control data in the complete table

9.3.3 Control Packet Arrival

This section explains the steps each node should perform upon arrival of the control packet (there is one control packet on each ringlet). Since the size of the control packet is a configurable system parameter, the data in one control packet can hold information from different source nodes. The steps to perform at each node when a control packet arrives are:

1. Delete information originating from "our" node. Shift other data, when existent, forward.
2. Copy the data from the control packet into the local table.
3. Schedule the control packet forwarding

Before describing these steps in detail, we define the following functions that are ringlet dependent. The other algorithms using these functions work for ringlets in both directions.
It should be noted that the functions explained below are a *possible* implementation, mainly intended for ease of understanding; it surely can be optimized (e.g. using a circular data buffer).

9.3.3.1 GetSourceID

When a control packet arrives with a certain offset value, and valid data, we need to know from which source this data originates. The following code return the source ID for data[index] (index is a number from 0 to ValidEntries). Recall that N is the number of nodes and M = N/2. RI is the ringlet identifier.

```
int getSourceID(int offset, int index){
   int pos = (offset+index)%(M*N);
   if (RI==0)
      return pos / M;
   else
      return (N-pos/ M)%N;
}
```

9.3.3.2 GetDestID

Similar to the previous function, the destination ID of data at position data[index]

```
int getDestID(int offset, int index){
```

```
   int pos      = (offset+index)%(M*N);
   int sourceID   = getSourceID(offset, index)
   if (RI==0)
     return ((pos%M)+sourceID+1)%N;
   else
     return (-(pos%M)+sourceID-1+N)%N;
}
```

9.3.3.3 Handle Control Packet Arrival

This function is the main packet arrival function; it calls the functions corresponding to the previously discussed steps:

1. Delete information originating from "our" node. Shift other data, when existent, forward.
2. Copy the data from the control packet into the local table.
3. Schedule the control packet forwarding

```
void handleControlPacket(c802_17ControlPacket *pck)
{
   int thisNode = mac->atNode->getId();

   // ok, delete everything that I wrote last time
   pck->deleteAllFromMe(thisNode,&nodeIsReady);

   // update our local table:
   pck->updateTable(localTable);

   // schedule the forwarding
   scheduleEvent(controlHoldTime,FORWARD_EVENT);
}
```

9.3.3.4 deleteAllFromMe

This function checks to see if the information at data [0] comes from nodeID. When this is the case, then it deletes all entries from nodeID. The boolean value "ready" is set to true if nodeID is ready, which means it has sent all its information and should not add any information to the control packet.

```
void deleteAllFromMe(int nodeID, bool *ready){
   *ready=false;
   int sourceID          = getSourceID(0);
   int remainingInRow    = M-(offset%M);
   int todel             = min (remainingInRow, validEntries);
   if ((sourceID==nodeID) && todel){
     if (remainingInRow==todel) *ready=true;

     memmove((void*)payload,(void*)(payload +todel),
             sizeof(entry)*(maxEntries-todel));
     offset      = (offset+todel) %(M*N);
     validEntries   = todel;
   }
}
```

An example of this function is shown in Figure 9.9. On the left we have the situation before and on the right the situation after the function execution. As before N=12 and M=6. Furthermore, we assume that the control packet enters at node 2.

Node 2 now sees that the first entry in the control packet (index 14) originates from itself, and that there are a total of 4 entries from node 2. Node 2 now removes these entries from the control packet, since this data has completed one full round, and shifts the remaining data forward. Node 2 has no more data to send in this round, and therefore sets the boolean variable ready to true. The offset will change to 18, validEntries to 4.

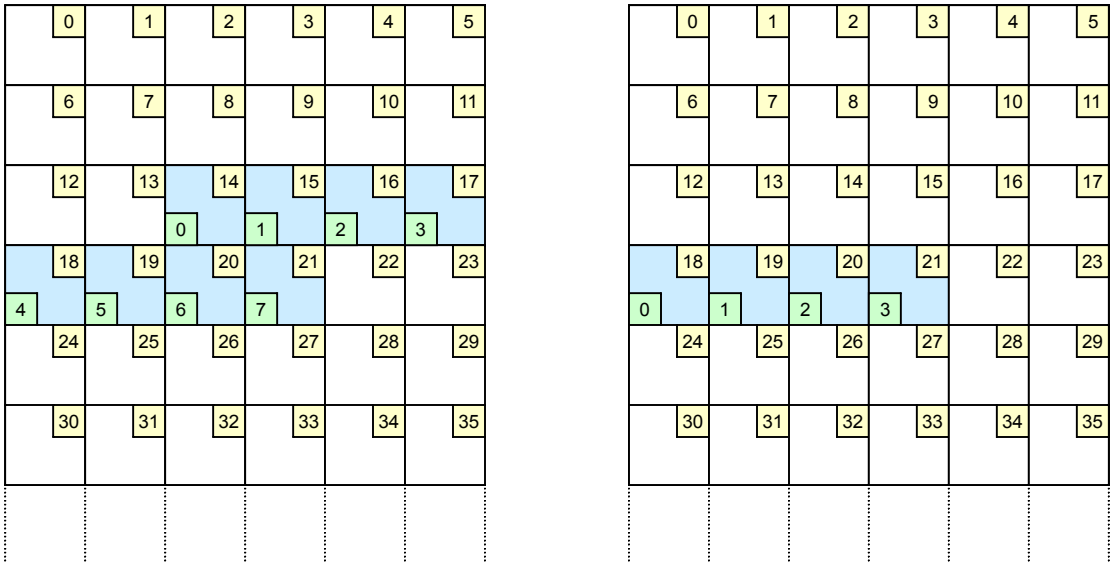


Figure 9.9 Example of the function "deleteAllFromMe"

9.3.3.5 UpdateTable

The function to update the table is straightforward, since all information in the data field is copied into the local table.

```
void updateTable(c802_17table *table){
   int i;
   for (i=0;i<validEntries;i++){
      int sourceID   = getSourceID(i);
      int destID     = getDestID(i);
      table->set(sourceID,destID,data[i]);
   }
}
```

9.3.4 Control Packet Forwarding

Forwarding is done after a fixed (though configurable) amount of time after receiving the control packet. Just before forwarding, we add our newest data to the packet if:

1. There is place available, and
2. It is my turn to add data

```
forwardControlPacket(){
   // nodeIsReady is set by deleteAllFromMe
   if (!nodeIsReady){
      int canAdd = ctrlPck->howMuchCanIadd(thisNode)
      while (canAdd){
         dest =  ctrlPck->getNextDestID()
         data = getQueueInfo(dest)
```

```
        localTable->set(thisNode,dest,data )
        ctrlPck->add(thisNode,data)
        canAdd--
      }
   }
   sendPacket(); // implementation dependent
}



int howMuchCanIadd(int nodeID){
   int leftOver = maxEntries-validEntries;
   if (leftOver<=0) return 0;        // FULL

   if (validEntries==0){             // complete Empty
      int remainingInRow = M-(offset%M);
      return min(leftOver,remainingInRow);
   }

   int firstFreePlace = validEntries;
   int sourceID   = getSourceID(firstFreePlace);

   if (nodeID==sourceID){
      int lastEntry  = validEntries+offset-1;
      int remainingInRow = M-(lastEntry%M)-1;
      if (remainingInRow == 0) remainingInRow = M;
      return min(leftOver,remainingInRow);
   } else
      return 0;
}



dataType getQueueInfo (int destID){
   // this function should read the buffer sizes
   // queue for destID (high and low priority)
   // and write the result in „dataType"

    implemenation dependent

}
```

The next function adds an entry with buffer info to the control packet:

```
void add(int nodeID, dataType &data){
   int firstFreePlace = validEntries
   payload[firstFreePlace] = data
   validEntries++
}
```

Example:
Figure 9.10 shows the control packet as it arrives at node 3. Node 3 detects that there are 4 valid entries and they all originate from node 3. This means that the information finished one round trip and node 3 deletes these 4 entries form the control packet (offset=22, validEntries=0).

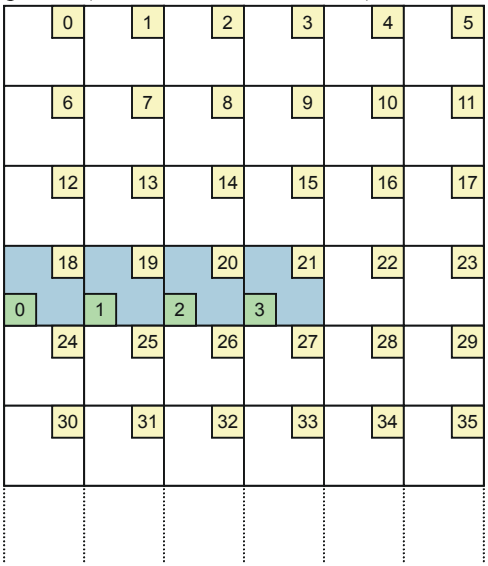


Figure 9.10 Control packet arriving at node 3

Node 3 however, has more data to advertise (flow 3→8 and 3→9) and puts this in the control packet and forwards the packet to node 4 (Figure 9.11, offset=22, validEntries=2).

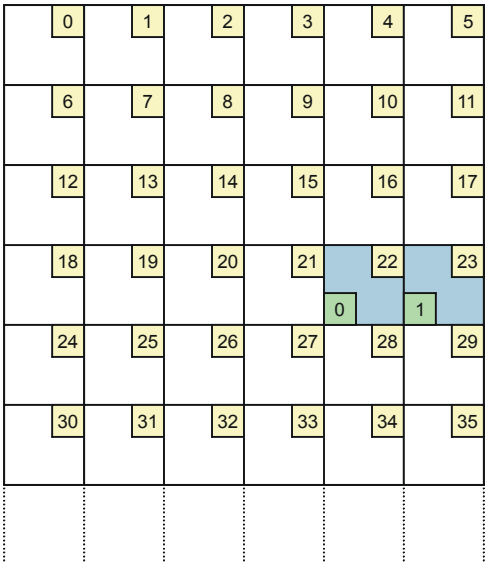


Figure 9.11 Control packet arriving at node 4

Node 4 sees that the first entry in the control packet originates from node 3; it copies this information into the local table and adds its own entries (Figure 9.12).

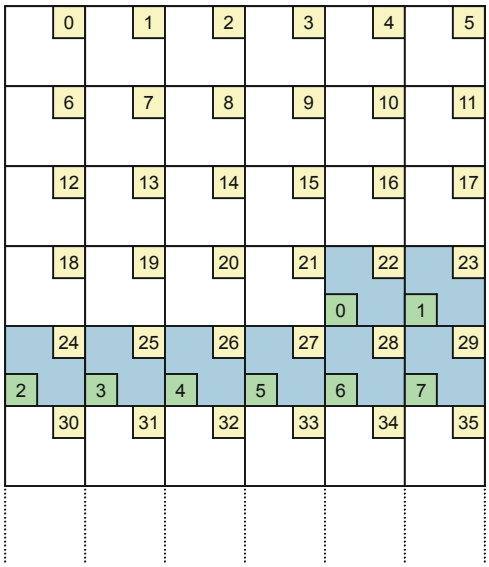
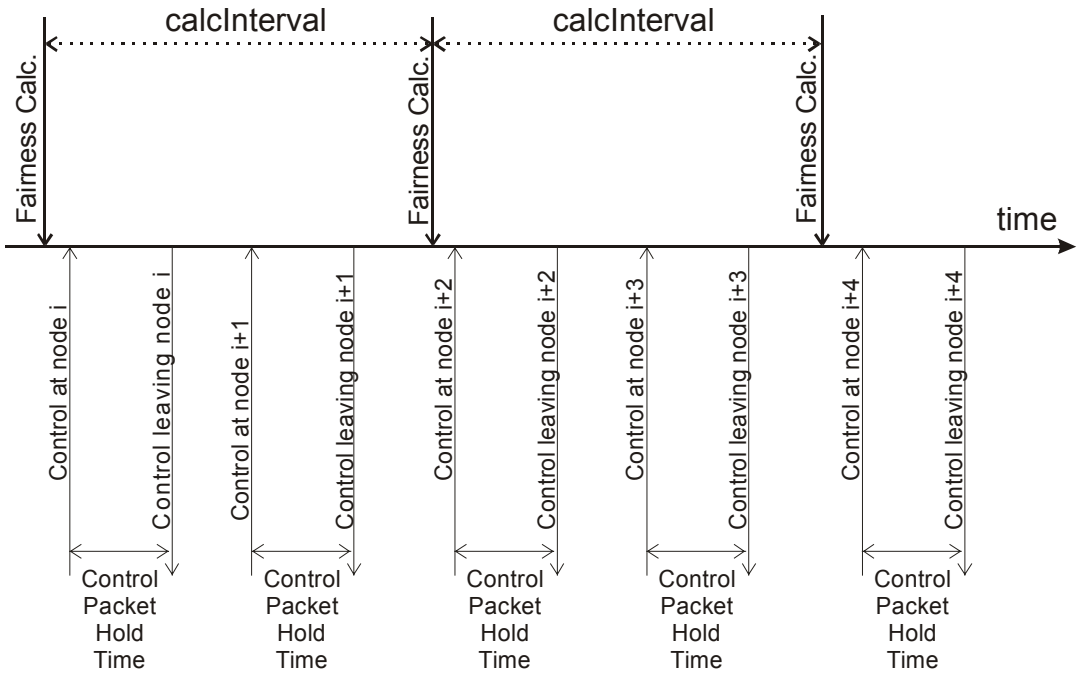


Figure 9.12 Control packet leaving node 4

9.3.5 Control Packet Loss Detection and Recovery

To be done…

9.4 Calculation Interval and Control Packet Timing



The circulation of the control packet and the calculation of the fair rates are two independent processes. The interval at which the nodes calculate their fair rates (calcInterval) should be a constant value and therefore should be triggered by the node itself. Although not crucial for correct operation, all nodes should have their calculation intervals synchronized, i.e., all nodes should calculate at the same time. The best way to achieve this remains for further study, but as said, the protocol is not very sensitive to this.

Another system parameter is the control packet hold-time. Together with the control packet size, this parameter directly controls the protocol overhead versus protocol performance. Note that the packet-hold time could be zero.