9. MAC fairness

9.1 Overview

9.1.1 Scope

This clause defines the fairness algorithm for RPR MACs. The MAC uses the algorithm to enforce fairness among stations on the ring. The RPR fairness algorithm RPR-FA handles the fairness eligible traffic (Class B and Class C traffic).

9.1.2 Goals and objectives

The fairness protocol has the following objectives:

a) Source-based weighted fairness—on any given segment on the ringlet, the available bandwidth is allocated to each station in proportion to its relative weight. For example, if every station has an equal weight, then the available bandwidth on the segment should be shared equally by all stations. On the other hand, if one station has a higher weight, the bandwidth allocated to that station should be in proportion to the station's weight divided by the sum of the weights of all the active stations.
b) Fast response time—In order to ensure maximum ring bandwidth utilization and to ensure that the protocol is responsive to instantaneous changes in traffic load, it must have a fast response time.
c) High bandwidth utilization on the ring—the protocol should be able to achieve very high levels of bandwidth utilization even under heavy load approaching 100% of the ring capacity.
d) Scalability—the protocol should be scalable and should be able to function predictably for all ringlet speeds and ring diameters allowed by this standard.

9.1.3 Relationship to other clauses

The RPR-FA is implemented within a control entity called the Fairness Control Unit (FCU) located in the MAC Control Sublayer, as described in Clause 5.

9.2 Acronyms

This clause contains the following acronyms:

FA	Fairness Algorithm
FCU	Fairness Control Unit
MTU	Maximum Transmission Unit
RTT	Round Trip Time
FCM	Fairness Control Message

9.3 Variables and terminology used

This clause contains the following definitions and variables in alphabetical order:

9.3.1 calculation round

The information collected in the information round is used by all stations in the calculation round. This round is where the fair rates are calculated by the fairness algorithm. Each station starts its new cycle after it has done its fair rate calculation in this round.

9.3.2 cycle

The intervals over which the fairness algorithm schedules and allocates the fair rates.

9.3.3 designated station

The station that initiates the information rounds, and therefore also the new cycles.

9.3.4 destinationTraffic

This array holds information about the local traffic load to each destination. It is a local array, i.e., each station holds one for each ringlet. The station fills this array during the information round, it is used during the calculation round.

9.3.5 estimator

Array holding the current estimated traffic to each destination

9.3.6 greedy mode

Source-destination flows not flowing over one or more bottleneck links are in greedy mode, i.e., free access in the current cycle. See also “reservation mode”.

9.3.7 information round

A round trip of the fairness control message just before the start of a new cycle, with the goal to collect information about the traffic demand on each link. This round is started by a designated station.

9.3.8 linkTraffic

This array holds information about the local traffic load on each link. It is a local array, i.e., each station holds one for each ringlet. The station fills this array during the information round, it is used during the calculation round.

9.3.9 numLinks

The number of links on the ringlet (equals numStation).

9.3.10 numStations

The number of stations on the ringlet (equals numLinks).

9.3.11 reservation mode

Flows not in greedy mode are in reservation mode, i.e., they flow over one or more bottleneck links.

9.3.12 table

Table is an array in the FCM that holds the following information for each link:

- demand: Total demand on the link
- remainingCap: Capacity available on the link

9.3.13 useSourceFairness

The fairness algorithm uses source fairness when this variable is set to true. Flow based fairness is used when this variable is set to false.

9.4 MAC fairness operation

The fairness algorithm implemented within the FCU consists of the following functions:

a) sourcing and consuming fairness messages
b) calculation of fair rates for each source destination pair
c) determining the state for each flow: reserved or free access

Each station is assigned a weight, which allows the user to allocate more ring bandwidth to certain stations as compared with other stations.

The FA uses a proactive method that assigns fair rates to each source-destination flow on a ringlet. It can be used for source-fairness as well as source-destination (or flow) fairness.
The algorithm uses three rounds of the FCM for each cycle. The first round is used for collecting the traffic demand for each source destination pair, and in the second round each station computes its own fair rate based on the information in the fairness message. Instead of using all flow information on the ringlet, the algorithm uses aggregate flows, keeping the fairness message small and the algorithm scalable. The third round is used to inform all stations about the available bandwidth that still can be used for greedy traffic.

The control information needed to accomplish this, flows in the same direction (i.e. ringlet) as the data flow, which simplifies the protocol in a single ring topology and any configuration of multiple rings.

9.4.1 FCM Processing

9.4.1.1 Fairness Control Message

The designated station creates fairness control messages. This message travels three rounds, thereby visiting each station on the ringlet three times. Apart from the normal packet header fields, the FCM contains a round counter (2 bits) and two arrays, each of size numLinks (the number of links on the ringlet), one for the total traffic demand on all links for fairness eligible traffic, and one array for the remaining capacity (total capacity minus provisioned traffic) on all links (Figure 1).

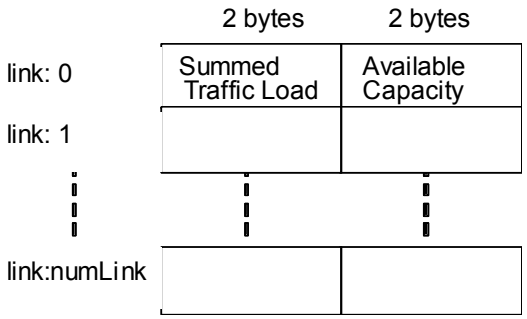

Figure 1. Table structure contained in the FCM

9.4.1.2 Overview

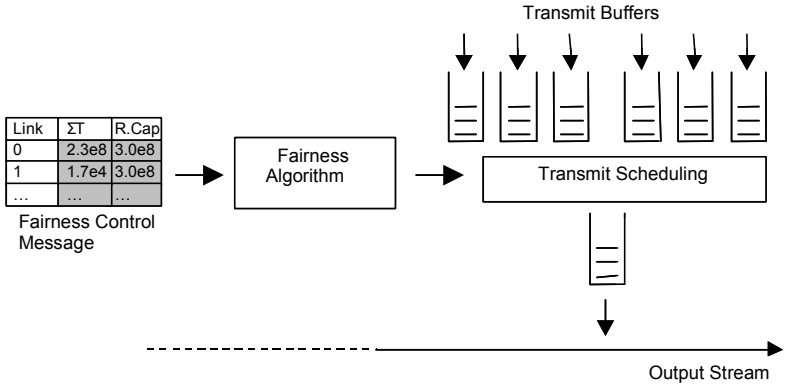

Figure 2. The fairness control message is input to the fairness algorithm

The designated station holds a timer for each ringlet that fires every cycle interval. At this event the designated station creates a fairness control message, initializes it and starts the first round of this message. In this "information gathering" round each station writes the amount of bytes that are available for its outgoing link, in the remaining capacity field. Additionally it also adds its own flows (information comes again from the waiting traffic demand) to the sum of all flows on all links. Once the control message arrives back at the designated station, the control message starts its second round where each station performs the fairness algorithm and immediately can start sending its fair share. In this second round the stations can modify the contents of the FCM. The third round is used to notify all stations about the amount of remaining capacity that can be used for free-access traffic. The control message is taken from the ringlet at the time it returns for the third time at the designated station.
Since the control message is relative small in size, it produces a small overhead even for short calculation intervals.

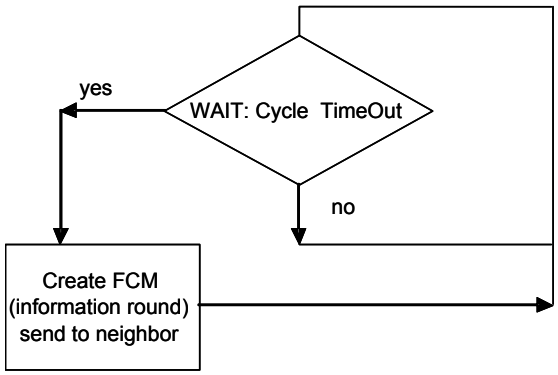

Figure 3. Cycle timeout for the designated station

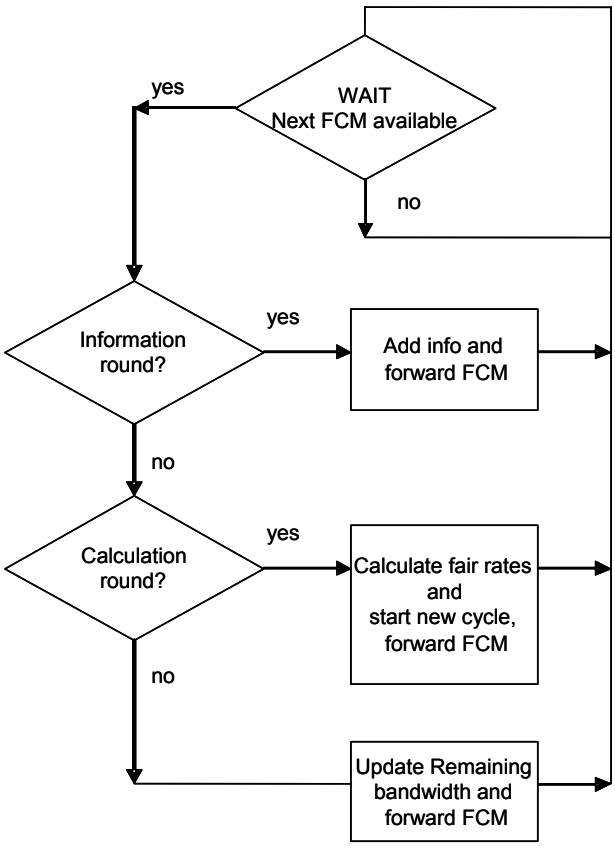

Figure 4. Flow diagram for non-designated stations

Each station starts its new cycle directly after calculation of the fair rates, i.e., stations can start transmission of the packets according to the just calculated fair rates.

The flow diagram for designated stations (Figure 5) is similar to the diagram for non-designated stations. The difference is that in the designated station the transition from one round to the other takes place. Additionally, the FCM will be deleted after the remaining capacity round.

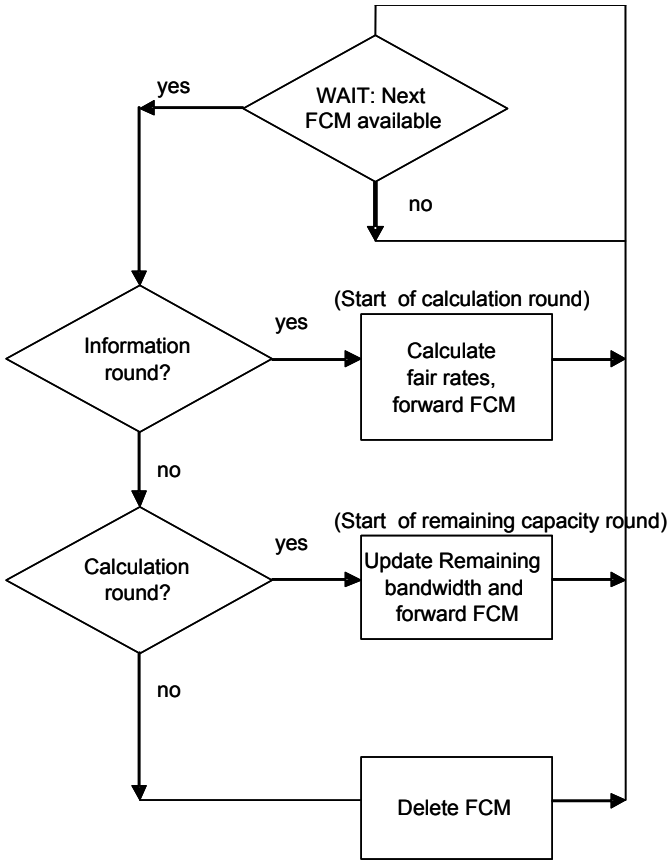

Figure 5. Flow diagram for the designated station

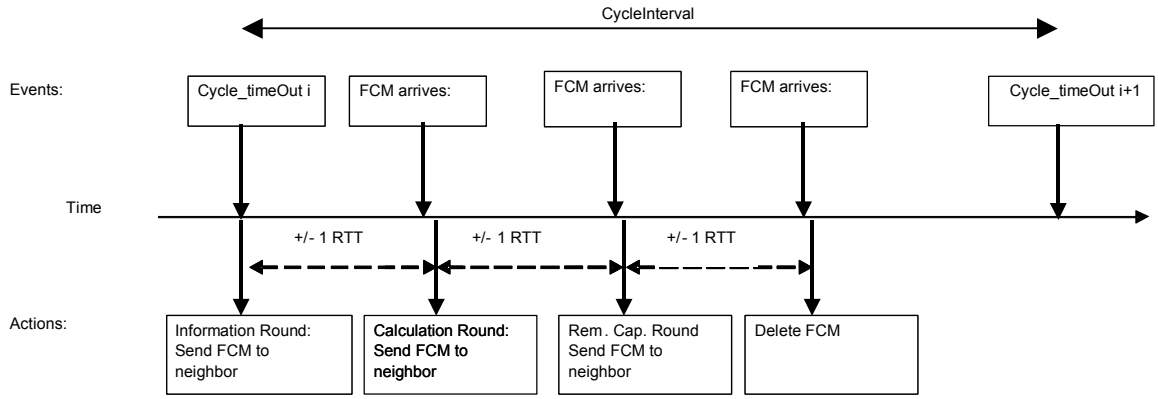

Figure 6. Time diagram for the designated station

9.4.2 Information collection round

The fairness algorithm is a proactive mechanism that uses the traffic demand to schedule the traffic in the next cycle. Ideally, the information about the waiting traffic demand comes from the MAC client. The rate will be estimated (Section 9.4.2.2), for clients unable to tell their traffic demand.

9.4.2.1 Getting the queued traffic demand

Upon request, the MAC-client notifies the MAC about the amount of bytes currently waiting in the destination queues. If queues are nearly or completely full, the MAC uses rate estimation, the topic of the next section.

9.4.2.2 Rate estimation

Two arrays are used to estimate the traffic flows: Estimated[] holds the current estimation to all destinations, and Measured[] holds the total amount of fairness eligible traffic sourced to each destination in the current cycle.
At the end of each cycle, all nodes perform the following update of the Estimated array: If there are no packets to transmit for a destination *d*, then Estimated[*d*] is set to Measured[*d*]. Else, we take the maximum of three values: the previous estimation, the measured value and a constant value B. This maximum multiplied by a constant A gives us the new estimation: Estimated[*d*] = A* max(Estimated[*d*], Measured[*d*], B).
The constants A and B control a trade-off between throughput and response time during transitions.

9.4.2.3 Weights

As long as there are no bottlenecks, weights are not used. When there are bottlenecks however, the estimated or real traffic demand is multiplied by the weight of each station. This value will be written in the FCM by each station in the information round.

9.4.2.4 Source and flow fairness

The fairness algorithm is a flow or source-destination fairness algorithm. Source fairness can be easily achieved with the same algorithm by limiting the sum of all flows leaving each station to the link capacity. These adjusted values are then written in the FCM.

9.4.3 Code

This section describes two functions: cycleTimeOut and handleFCM. CycleTimeOut is executed at each cycle timeout at the designated station. It generates a fairness control message and passes the message to the handleFCM function. This function is also executed at each station when a fairness message arrives.

```
void cycleTimeOut(){
   FairnessControlMessage *fcm = new FairnessControlMessage;
   fcm->round = -1;
   handleFCM(fcm);
}
```

```
void handleFCM(FairnessControlMessage *fcm)
```

```
{
   double *allowForDest;
   int i;

   if (designatedStation()){
      fcm->round++;
      if (fcm->round == 3){
           delete fcm;
           return;
      }
   }

   switch (fcm->round){
   case 0:            // information round
      infoRound(fcm);
      break;
   case 1:            // calculation round
      allowForDest=   new double[getSize()];
      makeFair(fcm,allowForDest);
      for (i=0;i<getSize();i++){
         setBitsAllow(i,allowForDest[i]);
      }
      delete allowForDest;
      break;
   case 2:            // update remaining capacity round
      for (i=0;i<getSize();i++){
         remainingCap[i] = fcm->table.available[i];
      }
      break;
   }

   // schedule the forwarding
   forwardFairnessControlMessage(fcm);
}
```

```
bool designatedStation(){
   // returns true if this station is the designated station
   // on the current ringlet, false otherwise;
}
```

```
int getSize(){
   // returns the number of stations on the ringlet;
}
```

```
void infoRound(…){
   // see Section 9.4.3.1
}
```

```
void makeFair(…){
   // see section 9.4.3.2
}
```

```
void setBitsAllow(int dst,double value){
   // Sets the amount of bytes that can be transmitted by this station
   // to destination dst.
}
```

```
void forwardFairnessControlMessage(FairnessControlMessage *fcm){
   // function that forwards the fcm to the next downstream neighbor as
   // quick as possible
}
```

9.4.3.1 Information Round

Each station in the information round executes the following function. The only argument is the fairness control message.

```
void infoRound(FairnessControlMessage *fcm)
{
   double sourceFairFactor       = 1.0;
   int i,dest,link;

   // set the available bandwidth to 100%
   fcm->table.available[atStationID()]=getCycleMaxLoad();


   // make a copy of the current fill sizes
   for (i=0;i<getSize();i++) {
      destinationTraffic[i] = min(getLoadForDestination(i), getCycleMaxLoad());
   }

   if (useSourceFairness)
      sourceFairFactor = calculateSourceFairFactor();

   // now loop through all links and add the amount of bytes
   link = downStreamLinkId();
   for (i=0;i<getSize();i++){

      // now find all destinations over "link"
      dest = stationIdAtEndOfLink(link);
      while (dest!=atStationID()){
         double Iwant = min(getCycleMaxLoad(),getLoadForDestination(dest));
         fcm->table.demand[link]  += Iwant * sourceFairFactor;
         linkTraffic[link]        =  fcm->table.demand[link];
         dest = next(dest);
      }
      link = next(link);
   }
}
```

```
double getCycleMaxLoad(){
   // return the maximum number of bits that can be transmitted on
   // the outgoing link in one cycle:
   // e.g. return cycleInterval * getLinkSpeed();
}
```

```
long getLoadForDestination(int dest){
   // see section 9.4.2.1
   double MAXpossible = getCycleMaxLoad();

   if (isQueueEmpty(dest))
      estimator[dest] = getBytesSend(dest);
   else
      estimator[dest] = A*max( estimator[dest], max(getBytesSend(dest), B));
```

```
   if (estimator[dest]>MAXpossible/2) estimator[dest]=MAXpossible/2;

   return estimator[dest];
}
```

```
bool isQueueEmpty(int dest){
   // returns true iff the MAC client has no fairness eligible traffic
}
```

```
int getBytesSend(int dest){
   // returns the amount of bytes sources to destination dest in the
   // current cycle
}
```

```
int stationIdAtEndOfLink(int lnk){
   // function that returns the id of the station at the end of link "lnk"
   // on the current ringlet
}
```

```
double calculateSourceFairFactor(){
   // this function computes a factor that is used by the fairness algorithm
   // to limit the flows of a single station. Limiting all flows from
   // one station to the link capacity, results in source fairness (in the
   // used fairness algorithm)

   double MAXpossible = getCycleMaxLoad();
   double ret    = 1.0;
   double total  = 0;
   int i;
   for (i=0;i<getSize();i++)
      total += min(MAXpossible,getLoadForDestination(i));

   if (total>MAXpossible)
      ret = MAXpossible / total;

   return ret;
}
```

9.4.3.2 Calculation Round

```
void makeFair(FairnessControlMessage *fcm, double *allowForDest){
   // This is the function that computes the fair rates for the station
   // where this function is being called. The FCM is input to this function,
   // allowForDest is the resulting array with the fair amount of bytes
   // for each destination.
   // Note that the contents of the fcm can be modified by this function.

   int strongestBottleNeckLink,i;
   bool *bottleNeckDone = new bool[getSize()];// array indicating whether or not
                                              // a bottleneck link is processed

   // initiliaze:
   for (i=0; i<getSize(); i++) {
      bottleNeckDone[i]    = false;
      allowForDest[i]    = 0;
   }
```

```
   do {
      // look for the strongest bottleneck where this station is involved...
      strongestBottleNeckLink=getStrongestValidBottleneck(bottleNeckDone,fcm);

      if (strongestBottleNeckLink!=-1){

         // yes we are involved in a bottleneck
         bottleNeckDone[strongestBottleNeckLink] = true;

         // ok, reduce all flows over this bottleneck
         // we start at the destination just over the bottleneck
         // and loop through all destinations from there on
         int toDest = stationIdAtEndOfLink (strongestBottleNeckLink);
         while (toDest != atStationID()){
            // do I have something for this destination?
            if (destinationTraffic[toDest]>0){
               double ratio        = fcm->table.ratio(strongestBottleNeckLink);
               double oldValue     = destinationTraffic[toDest];
               double newValue     = oldValue/ratio;
               destinationTraffic[toDest] = 0;
               allowForDest[toDest]       = newValue;

               // now update the table in the control message
               // and our local linkTraffic table accordingly.
               // all links between this station and toDest need
               // to be updated.
               double  diff = oldValue-newValue;
               int     link = downStreamLinkId();
               while (link != downStreamLinkIdAtStation(toDest)){
                  fcm->table.available[link]     -= newValue;
                  fcm->table.demand[link] -= oldValue;
                  linkTraffic[link]       -= diff;
                  link = next(link);
               }
            }
            toDest = next(toDest);
         }
      }
   } while (strongestBottleNeckLink!=-1); // as long as there are bottlenecks

   // copy remeainig traffic load since this
   // traffic is not involved in any bottleneck
   for (i=0; i<getSize(); i++) allowForDest[i] += destinationTraffic[i];

   delete bottleNeckDone;
}
```

```
int getStrongestValidBottleneck(bool *done, FairnessControlMessage *fcm){
   // Returns the id of the strongest bottleneck, that is not yet
   // processed (done).
   // Returns -1 if no such bottlenecks exists.

   int i,ret =-1;
   for (i=0;i<getSize();i++)
      if (!done[i] && fcm->table.isBottleNeck(i) &&
         ((ret==-1) || (fcm->table.ratio(ret)<fcm->table.ratio(i))))
              ret = i;
   return ret;
}
```

```
int downStreamLinkId(){
   // Returns the downstream link id at the current station on the
   // current ringlet
}
```

```
int downStreamLinkIdAtStation(int s){
   // Returns the downstream link id at station "s" on the current ringlet
}
```

```
int atStationID(){
   // Returns the station ID of the station
}
```

```
bool table::isBottleNeck(int lnk){
   // We have a bottleneck if the demand is larger than what is available,
   // and there is a positive non-null demand:
   return (demand[lnk]>0) && (demand[lnk]>available[lnk]);
}
```

```
double table::ratio(int lnk){
   // Returns the ration demand/available for the specified link
   // To avoid division be zero, a very large constant "BIG" is
   // returned if available equals zero.

   if (available[lnk]==0) return BIG;
   else return demand[lnk]/available[i];
}
```

9.5 Example

This section gives an example of the operation of the algorithm. For simplicity a single ringlet is used with one priority class and only 4 stations. Furthermore, the following assumptions are made:

- 100 units (e.g. bytes) can be transmitted on each link, in one cycle
- Station 0 is the designated station
- Weights are all 1
- The traffic demand from and to each station is shown in Figure 7.

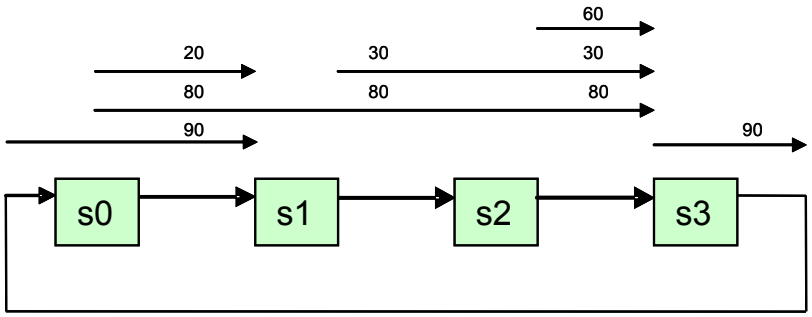

Figure 7. Traffic demand

9.5.1 Information round

The information round starts at the designated station 0 after receiving the cycle timeout. It creates a new FCM and adds all of its flows to the summed traffic fields. The remaining capacity is set to full capacity (100).

Link	Summed Traffic	Remaining Capacity
0	20+80=100	100
1	80	100
2	80	100
3	-	100

Station 0 forwards this FCM to station 1, which also adds it own flows to the summed traffic fields.

Link	Summed Traffic	Remaining Capacity
0	100	100
1	80+30 =110	100
2	80+30 =110	100
3	-	100

FCM leaving station 1

Similar for station 2 and station 3:

Link	Summed Traffic	Remaining Capacity
0	100	100
1	110	100
2	110+60 = 170	100
3	-	100

FCM leaving station 2

Link	Summed Traffic	Remaining Capacity
0	100+90 = 190	100
1	110	100
2	170	100
3	90	100

FCM leaving station 3

9.5.2 Calculation round

A new round is started whenever the designated station receives the FCM from its upstream neighbor, in the example the new round will be the calculation round. This is the round where each station calculates its own fair rates and immediately can start to transmit these fair rates.

The highest bottleneck where station 0 is involved is at link 0. All flows leaving station 0 over this bottleneck will be reduced by a factor 100/190. For flow 0->1 with a demand of 20, the assigned value will become 20*100/190 = **10.5**.
The FCM has to be updated to reflect this: The summed traffic fields involved should be decreased by 20, which is in this case only the field for link 0. Since 10.5 is assigned to this flow, the remaining capacity fields should be decreased by 10.5.
The new FCM:

Link	Summed Traffic	Remaining Capacity
0	190-20=170	100-10.5=89.5
1	110	100
2	170	100
3	90	100

Station 0 still has a flow over a bottleneck, which is still link 0. Flow 0->3 with a demand of 80 will get a value of 80 * 89.5 / 170 = **42.1.**

The new FCM:

Link	Summed Traffic	Remaining Capacity
0	170-80 = 90	89.5-42.1=47.4
1	110-80 = 30	100-42.1=57.9
2	170-80=90	100-42.1=57.9
3	90	100

Equals:

Link	Summed Traffic	Remaining Capacity
0	90	47.4
1	30	57.9
2	90	57.9
3	90	100

FCM leaving station 0

Station 1 follows the same procedure: highest bottleneck is at link 2, the assigned value will be 30 * 57.9 / 90 = **19.3.**

Link	Summed Traffic	Remaining Capacity
0	90	47.4
1	30-30=0	57.9-19.3=38.6
2	90-30=60	57.9-19.3=38.6
3	90	100

FCM leaving station 1

Station 2: flow gets 60 * 38.6 / 60 = **38.6**

Link	Summed Traffic	Remaining Capacity

0	90	47.4
1	0	38.6
2	60-60=0	38.6-38.6=0
3	90	100

FCM leaving station 2

Station 3: flow gets 90 * 47.4/90 =**47.4**

Link	Summed Traffic	Remaining Capacity
0	90-90=0	47.4-47.4=0
1	0	38.6
2	0	0
3	90-90=0	100-47.4=52.6

FCM leaving station 3

All flows are now assigned, as can be seen in the following figure:

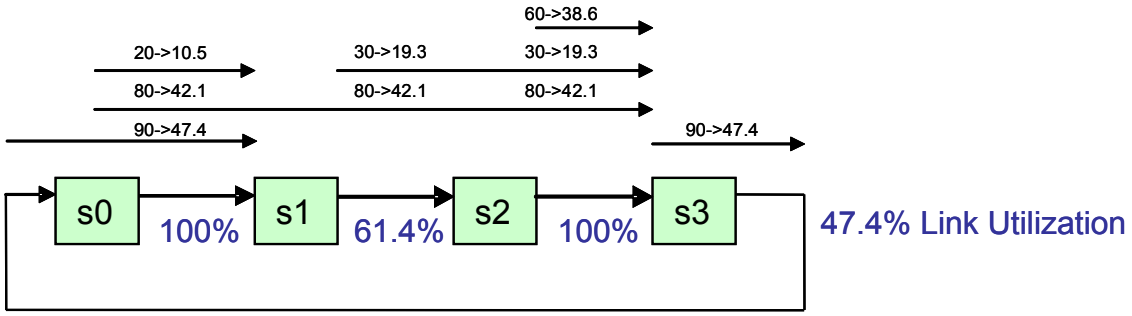

Figure 8. Result after the calculation round

9.5.3 Remaining Capacity Round

The purpose of this round is to inform all station on the ringlet about the amount of capacity available on all links. This is used for greedy traffic.

Link	Summed Traffic	Remaining Capacity
0	0	0
1	0	38.6
2	0	0
3	0	52.6

When the FCM returns back to the designated station, the FCM will be deleted. The timer in the designated station will trigger the start of the next cycle.