Congestion Control and Traffic Management in ATM Networks
Recent Advances and a Survey

Raj Jain

The objective of *traffic management* is to ensure that each connection gets the quality of service it was promised.

The use of fixed size (48 byte payload) of cells produces determininstic service times and reduces variance of delay.

Traffic management is made more difficult
- by intermittent periods of heavy load and
- by unpredicable loads.

Congestion control is the most essential component of traffic management.

ATM networks are connection oriented.
- Intermediate switches should be informed of demand and connection setup time.
- Call refusal and/or call rerouting can be used in ATM traffic management

ATM QoS parameters

Peak Cell Rate (PCR) -

The maximum instantaneous rate at which cells will be injected by a traffic source

Sustained Cell Rate (SCR) -

The average cell rate when measured over a "long" interval.

Cell Loss Ratio (CLR) -

The fraction of cells lost in the network due to transmission errors (typically very small in optical networks) and being dropped because of congestion. The probability of a cell being dropped can be reduced by setting the CLP (Cell loss priority) bit in the cell header to 0.

Cell Transfer Delay (CTD) -

The time it takes a cell to traverse the ATM network from entry to exit. It is comprised of propagation delays, queuing delays and actual transmission time.

Cell Delay Variation (CDV) -

A measure of the variance of the CTD, the CDV can be measured in various ways. One way is to compute the difference between the *x* percentile delay (where *x -> 100)* and the minimum delay

Cell Delay Variation Tolerance (CDVT) -

The maximum tolerated difference between the nominal interarrival time and the actual interarrival time of arriving cells. (Jain seems to apply this to injected cells)

Maximum Burst Size (MBS) -

The size of the maximum continuous burst of cells that can be sent.

Burst Tolerance (BT) -

BT = (MBS - 1) (1 / SCR - 1 / PCR)

Minimum Cell Rate (MCR) - Minimum rate at which cells will be injected.

MBS and MCR appeared in UNI 3.1

Traffic classes

CBR	PCR, CDVT, Peak to Peak CDV, Max CTD , CLR
VBR(RT),	PCR, SCR, MBS, CDVT, Peak to Peak CDV, Max CTD, CLR
VBR(nRT)	PCR, SCR, MBS, CDVT, Mean CTD, CLR
ABR	PCR, CDVT, MCR, CLR, Congestion feedback
UBR	PCR, CDVT,

Congestion managment mechanisms

Capacity planning and network design --- Very long term
Connection admission control
Dynamic routing (at the connection level in ATM)
End-to-end feed back
Link-by-link feed back
Buffering --- Very short term

The Generalized Cell Rate Algorithm (GCRA)

Leaky bucket based
Non-compliant cells can be dropped or have CLP bet set to 1.
For CBR, non compliance is governed by GRCA(1/PCR, CDVT)
Interarrival times of compliant cells must be >= [1/PCR - CDVT]
For other traffic classes it is: GRCA(1/SCR, BT)
Interarrival times of compliant cells must be >= [1/SCR - BT]

Original congestion Feedback Mechanisms in ATM

UNI 3.0
Generalized Flow Control (GFC) -

4 bits in the UNI cell header -- should be viewed as a bug in the standard

Explicit Forward Congestion Indication (EFCI) -

The second bit of the PTI in user cells is used as a congestion indicator. Recipient of a Cell with EFCI set should be able to notify the sender to reduce the rate.

Subsequent congestion management recommendations

Fast resource management - France Telecom

RM cell required before each burst

Delay - based rate control. - Fujitsu -

Round trip time measurements used to adjust sending rate.

Backward ECN - N.E.T

RM cells used as choke packets to require packet rate be cut in half. Packet rates are doubled each period until P/SCR rate reached.

Early Packet Discard - SUN

Optimized for AAL5... discard ALL cells of a single AAL5 frame

Link Window based flow control - Tzeng and Siu

Window based flow control on EACH link with EFCI based end to end control -- guaranteed 0 drops.

Fair queing with Rate and Buffer Feedback (Xerox / Cisco)

Fair share rates are determined by switches and used to update RM cells by lowering the explicit rate if need be).

Credit based approach

Per-link and Per-VC window based flow control.

etc. etc. etc.

Selection critera for traffic management mechanisms

Scalability -

Optimality - Optimality -

max-min allocation or weighted max-min allocation are commonly used:

http://www.cs.berkeley.edu/~kfall/EE122/lec26/sld026.htm

Computing maxmin fair allocation

Given a set of traffic classes with ordered demands *d1, d2, .. ,dn* and a link capacity C, unweighted maxmin fair allocation is O(n).

All maxmin fair algorithms employ reduction in which

the most constrained flows are granted capacity first.

the selected flow is then removed from the collection and the next most constrained flow is selected

the algorithm ends when all flows are processed.

Local maxmin fair allocation given ordered demands for capacity.

```
#include <stdio.h>
#include <math.h>

double demands[] = {200, 260, 400, 500};
#define COUNT (sizeof(demands) / sizeof(demands[0]))
double allocs[COUNT];
#define CAPY 1000
double left = CAPY;
double share;
double sum;

int main()
{
   int i;

/* If demands are ordered maxmin can be O(N)        */
/* Fair “share” is remaining capy / remaining flows */

   for (i = 0; i < COUNT; i++)
   {
      share = left / (COUNT - i);

   /* The minimum of a flows fair share and its demand is allocated */

      if (demands[i] <= share)
         allocs[i] = demands[i];
      else
         allocs[i] = share;

   /* Subtract amount actually allocated from remaining capy */

      left = left - allocs[i];
      sum += allocs[i];
      printf("%3d %8.0lf \n", i, allocs[i]);

   }
   printf("allocated %8.0lf \n", sum);
}

==> maxmin
  0      200
  1      260
  2      270     <--- did not receive entire demand
  3      270
allocated     1000
```

Computing weighted maxmin fairness -

Each traffic class *i* has an associated weight *wi*

The fair share of a class is *wi / (sum(wj)) * capacity*

If all weights are equal the problem reduces to unweighted maxmin fairness and *share = capacity / classes*

For example:

```
double demands[] = {400, 200, 1000, 400};
double weights[] = {2.5, 4, 0.5, 1};
```

Here weights sum to 8 and capacity is 1400. Hence a basic share is 1400 / 8 = 175, and the fair share assignment of capacity is: {2.5 * 175, 4 * 175, 0.5 * 175, 175} = {437.5, 700, 87.5, 175}

After the first round of allocations the allocated bandwidth is

```
(gdb) print allocs
$5 = {400, 200, 87.5, 175}
```

and the excess (unused by flows 1 and 2) is 437.5 - 400 + 700 - 200 = 537.5

```
(gdb) print excess
$6 = 537.5
(gdb)
```

For the second round of allocations the sum of the weights is 1.5 and so the additional bandwidth available to flow 2 is 537.5 / 3 = 179.17 and to flow 3 is 2 * 537.5 / 3 = 358.33. Flow 3 already has 175 of its desired 400 and it leaves an excess of 175 + 358.33 - 400 or 133.33. This leaves flow 2 with 87.5+179.17+133.33 = 400.

The weighted maxmin allocation algorithm

```
double demands[] = {400, 200, 1000, 400};
double weights[] = {2.5, 4, 0.5, 1};

#define COUNT (sizeof(demands) / sizeof(demands[0]))
double allocs[COUNT];
#define CAPY 1400
double left = CAPY;
double share;
double sum;
double excess;

int main()
{
   int i;
   int j;
   double sumwt = 0.0;

/* Compute the sum of the weights */

   for (i = 0; i < COUNT; i++)
   {
      sumwt += weights[i];
   }

/* An O(N*N) algorithm is used.  Each trip through the outer loop */
/* fairly allocates remaining capacity to classes with remaining  */
/* demand according to the weight (higher is better) of the class */

   for (j = 0; j < COUNT; j++)
   {

   /* A "basic share" of the remaining capacity is the remaining*/
   /* capacity divided by the sum of the weights of the classes */
   /* that have remaining unsatisfied demand.  The amount       */
   /* that can be allocated is a "basic share * the weight      */
   /* assigned to the class.                                    */

      share = left / sumwt;
      excess = 0.0;
      sumwt = 0.0;
```

```
/* Each iteration of the inner loop examines a single class  */
/* Three cases are distinguished:                            */
/*    Demand already satisfied -> do nothing                 */
/*    Remaining demand exceeds current fair share -> add     */
/*                current fair share to allocation and class  */
/*                weight to sum of classes wanting more       */
/*    Current fair share exceeds remaining demand -> satisfy */
/*                demand and remember unneeded capy in excess */

   for (i = 0; i < COUNT; i++)
   {

   /* Ignore classes whose allocation has been fully met */

      if (allocs[i] < demands[i])
      {
         double wanted = demands[i] - allocs[i];

      /* If demand exceeds fair share, allocate only fair share  */
      /* and add weight of this class  to the sum of the weigths */
      /* of classes still contending for resources               */
      /* those still contending for resources                    */

         if (wanted > weights[i] * share)
         {
            allocs[i] += weights[i] * share;
            sumwt += weights[i];
         }

      /* If demand can be fully satisified,  allocated   */
      /* reqd amount and assign any leftover to the excess pool */

         else
         {
            excess += weights[i] * share - wanted;
            allocs[i] = demands[i];
         }
      }
   }
   left = excess;

/* Exit outer loop if all are satisfied or out of gas */

   if ((sumwt == 0) || (excess == 0))
      break;
}
```

```
==> wmaxmin
 0 -      400
 1 -      200
 2 -      400
 3 -      400
```

Fairness index

Suppose Oi is the optimal allocation for class i
Suppose Ai is the actual allocation for class i.
Then Ni = Ai / Oi is the normalized allocation.
Fairness = sum(Ni) * sum(Ni) / n * sum(Ni * Ni)
The maxmum value of fairness = 1.0 which is perfectly fair.
The minimum value -> 0 as the number of flows -> infinity and perfectly unfair allocation is used.

Ni	Ni*Ni
1.1	1.21
0.9	0.81
0.8	0.64
1.2	1.44
4	4.1
16	16.4
0.98 Fairness	

4	16
0	0
0	0
0	0
4	16
16	64
0.25 Fairness	

Congestion Control with Explicit Rate Notification

A. Charny, D. Clark, R. Jain

Time Scale Analysis and Scalability Issues for Explicit Rate Allocation in ATM Networks

A. Charny, K. Ramakrishnan, A. Lauck

Computing global maxmin fair allocation

S = flows in the network
L = links in the network

Cj = Capacity of Lj
Nj = number of flows on Lj

L1 = links for which b1 = min{Cj / Nj} is achieved.

Consider the Jain example:

Flows: S1, S2, S3, S4
Links: L1, L2, L3

S1 traverses L1,
S2 traverses L1,
S3 traverses L1, L2
S4 traverses L2, L3

C1/N1 = 150/3 <--- primary bottleneck
C2/N2 = 150/2
C3/N3 = 150/1

==> Assign b1 (50) to flows S1, S2, S3
==> Remove L1 and S1, S2, S3 from the network
==> Subtract the bandwidth allocated to removed flows from the links they traverse
==> Repeat the procedure until all links or all flows have been removed.

Computing maxmin fairness in a *network*.

The previous algorithms we examined computed maxmin fair allocation on an isolated link given (weighted varying traffic demands).

This algorithm assumes:

Infinite traffic demand for each flow
Fixed capacity on each link in the network
and allocates bandwidth in a maxmin fair way to each flow.

```
#define LINKS 4
#define FLOWS 6

/* Links used by each flow ... */

int routes[FLOWS][LINKS] =
{
   {1, 1, 1, 1},
   {0, 1, 1, 1},
   {0, 0, 1, 1},
   {1, 1, 1, 0},
   {1, 1, 0, 0},
   {1, 0, 0, 0}
};

double share[FLOWS][LINKS];

int linkflags[LINKS];

double n[LINKS];   // Current number of flows using link
double b[LINKS];   // Equal share bandwidth on link
double minb;       // Minimum of all b[] values
int    minid;      // Index of minimum b value
int    flowsleft = FLOWS;

#define MAX_CAPY 100

// double capy[LINKS] = {100.0, 100.0, 100.0, 100.0};
double capy[LINKS] = {50.0, 80.0, 60.0, 20.0};
```

```
int main()
{
   int f;
   int l;
   int i;

/* Each iteration of the outer loop reduces the network */
/* by exactly one link and at least one flow.. We think */
/* we are done when no flows are left to be reduced     */

   for (i = 0; i < LINKS; i++)
   {
      if (flowsleft == 0)
         break;

   /* Compute the number of flows using each links     */
   /* because of the reduction process this can change */
   /* Each iteration                                   */

      memset(n, 0, sizeof(n));
      for (f = 0; f < FLOWS; f++)
      {
         for (l = 0; l < LINKS; l++)
         {
            n[l] += routes[f][l];
         }
      }
```

```
/* Find the bottleneck link... This is by definition  */
/* the link whose current capacity divided by current */
/* number of flows is minmal.. If there is a tie just */
/* pick the first one.   The linkflags[] array keeps  */
/* track of the links that have been removed.         */

   minb =  MAX_CAPY;
   for (l = 0; l < LINKS; l++)
   {
      if (linkflags[l] == 0)
      {
        b[l] = capy[l] / n[l];
        if (minb > b[l])
        {
           minb = b[l];
           minid = l;
        }
     }
    }
```

```
    /* Remove the bottleneck link and all flows that */
    /* traverse it from the network                  */

       linkflags[minid] = 1;
       for (f = 0; f < FLOWS; f++)
       {

       /* Consider only flows crossing the bottleneck */

          if (routes[f][minid])
          {
             flowsleft -= 1;

          /* To remove a flow we must alter each link */
          /* the flow uses. The flows share on each   */
          /* link is set to the bottleneck bandwidth  */
          /* and the remaining capacity of the link   */
          /* reduced.. removal of the flow is done    */
          /* by setting route[f][l] back to 0         */

             for (l = 0; l < LINKS; l++)
             {
                if (routes[f][l])
                {
                   routes[f][l] = 0;
                   share[f][l] = minb;
                   capy[l] -= minb;
                }
             }
          }
       }
   }

   for (f = 0; f < FLOWS; f++)
   {
      for (l = 0; l < LINKS; l++)
      {
         printf("%8.0lf ", share[f][l]);
      }
      printf("\n");
   }
}
```

```
 ==> dmaxmin
       7         7         7         7
       0         7         7         7
       0         0         7         7
      14        14        14         0
      14        14         0         0
      14         0         0         0
-------------------------------------
      50        80        60        20   Link capacity


 ==> dmaxmin
      10        10        10        10
       0        10        10        10
       0         0        30        30
      10        10        10         0
      10        10         0         0
      20         0         0         0
-------------------------------------
      50        40        60        80
```

Distributed maxmin fairness and ATM ABR

For ABR a congestion feedback mechanism is used

Host role -

After every *k* cells, a resource management (RM) cell must be transmitted.
The RM cell contains the rate the sender would *like* to transmit at the moment.
This value is called the *ER* explicit rate
Congested switches may reduce the *ER* before forwarding the RM cell
When it reaches the destination the RM cell is reflected back to the sender and may have the *ER* lowered on the reverse path as well.
On receiving the RM cell the sender is expected to set its *ACR* actual cell rate to the value specified as the *ER*.

Switch role -

For each link the switch maintains some sort of ARMA cell rate for each connection

The switch partitions the connections into two classes

not-yet regulated -	rate higher than switch's advertised rate
regulated -	rate less than or equal to advertised rate

The switch computes the advertised rate as:

$$\frac{\text{capacity-consumed-by-not-yet-regulated flows}}{\text{number of not-yet regulated flows}}$$

This may cause some previously restricted flows to be unrestricted.
A second recalculation suffices to properly partition the flows.

The “advertised rate” is used to adjust the *ER* in the RM cells.

It is shown that this approach will converge to globally maxmin fair allocation.