به نام خدا

دانشگاه بو علی سینا

گزارش کار پیاده سازی

عنوان مقاله

AFCD: An Approximated-Fair and Controlled-Delay Queuing for High Speed Networks

استاد راهنما : دکتر محمد نصیری

نویسندگان : محمد پیشدار (کارشناسی ارشد فناوری اطلاعات)

نرگس رضایی (کارشناسی ارشد فناوری اطلاعات)

خلاصه ی مقاله :

AFCD الگوریتمی است که در سال 2013 در شبکه های high speed درسطح روتر و با استفاده از مدیریت صف ارائه شد که با استفاده از تخمین نرخ ارسالی جریان ها کار کرده و در صدد رسیدن به اهداف زیر بوده است (انگیزه مقاله):

- **Fairness**
- **Minimal Queuing Delay**
- **Acceptable Link Utilization**
- **Simple Implementation**

قبل از ارائه این الگوریتم ، الگوریتم های زیادی در سطح روتر پیاده سازی شده اند مثل AFQ,SBF,CODEL و . . . . اما هر کدام از این الگوریتم ها تنها یکی از اهداف ذکر شده را تامین می کردند. به عنوان مثال الگوریتم CODEL تنها تاخیر کم صف را تامین میکند و الگوریتم AFQ تنها fairness را دارد. الگوریتم AFCD برگرفته شده از دو الگوریتم AFQ , CODEL می باشد .

AFQ : **این الگوریتم یک عدالت نسبی را بین جریانهای موجود فراهم میکند . به عبارتی باعث می شود که جریان ها سهم یکسانی را از پهنای باند داشته باشند.در این الگوریتم تصمیم گیری برای دور انداختن پکت ها بر اساس نرخ ارسالی جریان ها صورت می گیرد.**AFQ **برای تخمین زدن نرخ ارسالی جریان ها از** SHADOW BUFFER **و** FLOW TABLE **استفاده می کند .** FLOW TABLE **برای نگه داشتن تعداد پکت های هر جریان استفاده می شود . اگر** mi **را مقدار ترافیک جریان** i **دریک** interval **در نظر بگیریم ، انگاه خواهیم داشت :**

$$m_{fair} \leftarrow m_{fair} + \alpha(Q_{old} - Q_{target}) - \beta(Q - Q_{target}) \quad (1)$$

مقدار جریان i **رابر حسب** mi **حساب می کند وهمچنین مقدار** r fair **را نیز بر اساس** m fair **حساب می کند . احتمال دور انداختن یک پکت که متعلق به جریان** i **می باشد :**

$$D_i = (1 - r_{fair}/r_i)_+ \quad (2)$$

CODEL : **این الگوریتم سعی در ایجاد تاخیر صف کم و حل مشکل** bufferblot **دارد. این االگوریتم تاخیر صف هر چکت را محاسبه کرده و چنانچه این تاخیر از تاخیر** target **بیشتر شود برای آخرین** interval**، آنگاه پکت حذف شده**

و قانون کنترل codel **برای زمان حذف بعدی تنظیم می شود . اگر تاخیر پکت کمتر از تاخیر** target **شود انگاه قانون کنترل** codel **حذف کردن پکت ها را متوقف می کند .**

: AFCD

در این الگوریتم از دو تابع DEQUEUE **و** ENQUEUE **استفاده شده است . وقتی پکت می رسد بسته به اینکه متعلق به کدام جریان است جداول** shadoe **و** buffer **به روز میشوند .این عملیات در داخل تابع** enqueue **انجام گرفته و پس از ان تصمیم گیری برای حذف کردن یک بسته در داخل** dequeue **آغاز می شود .**

Fig. 1: Functional block diagram of AFCD queuing

بر اساس flow table **مقدار** mi **به صورت زیر محاسبه می شود :**

$$m_i \leftarrow flow\_table[i] \tag{3}$$

$$m_f \leftarrow \frac{SHADOW\_BUFFER\_SIZE}{flow\_count} \tag{4}$$

$$m_f(t2) \leftarrow m_f(t1) + \alpha(Q(t1) - Q_{target}) - \beta(Q(t2) - Q_{target}) \tag{5}$$

داخل تابع dequeue **یک** target-delay **داریم که همانند الگوریتم** codel **با استفاده از اپراتورهای شبکه تنظیم می شود . در** afcd **برای هر جریان یک** target-delay **به صورت زیر محاسبه می شود:**

$$target\_delay_i \leftarrow target\_delay \times \left(\frac{m_i}{m_f}\right)^{-3} \tag{6}$$

اگر مقدار پهنای باندی که جریان از ان استفاده می کند کمتر از پهنای باند عادلانه باشد در این صورت مفدار target-delay **حاصل از ان جریان بیشتر از** target-delay**شبکه میشود .**

شروع پیاده سازی در NS3

گام اول (افزودن کلاس Afcd **به ماژول** Internet**)**

شبیه ساز Ns3 **شبیه سازی است که از ماژول ها ی مختلف تشکیل شده است که هر کدام از آنها شامل فایل ها ی با موضوعات مشابه می باشد .**

به دلیل اینکه AFCD **در دسته ی مدیریت صف در روتر ها می باشد . در ماژول** Internet **در پوشه ی** SRC **کلاس** Afcd-queue.cc **و** Afcd-queue.h **را اضافه کردیم .**

روش افزودن : ابتدا باید نام کلاس جدید را در فایل wscript **مربوط به ماژول اینترنت ثبت کرده و سپس .** ns3 **را مجددا** build **و** waf **را اجرا کنیم تا کلاس های جدید به** Ns3 **شناسانده شوند .**

سپس وارد مرحله ی تعریف کلاس Afcd **می شویم .**

طبق خلاصه ی مقاله که در ابتدا آورده شد برای پیاده سازی Afcd **نیاز به توابع و آرایه ها و متغییر های زیر داریم**

ابتدا Shadow Buffer **و** Flow Table **را به صورت زیر تعریف کرده ایم**

```
#define  SHADOW_BUFFER_SIZE 200

struct packetDATA{int isfull;
                  uint32_t srcip;
                  uint32_t desip;};


packetDATA SHADOW_BUFFER[SHADOW_BUFFER_SIZE];

 uint32_t FLOW_TABLE[SHADOW_BUFFER_SIZE];
```

توابع مهم پیاده شده در کلاس AFCD :

```
uint32_t iphashing_key(uint32_t a, uint32_t b)
{
  int primer1=7;

  uint32_t key=((a+b)*primer1) % SHADOW_BUFFER_SIZE;
  std::cout<<a<<" "<<b<<" "<<key<<std::endl;
  return key;
}
```

این تابع دو عدد را به عنوان ورودی گرفته و یک کلید Hash **را بر می گرداند (با توجه بو اندازه ی** Shadow Buffer**)**

```
uint32_t iphashing_step(uint32_t a, uint32_t b)
{
  int primer2=5;
  uint32_t step=((a+b)*primer2) % SHADOW_BUFFER_SIZE;
  return step;
}
```

این تابع در صورت وجود برخورد در Hash گام بعدی را محاسبه می کند

```
static uint32_t AfcdGetTime (void)
{
  Time time = Simulator::Now ();
  uint64_t ns = time.GetNanoSeconds ();

  return ns >> Afcd_SHIFT;
}
```

این تابع زمان را به صورت نانو ثانیه باز می گرداند

```
void
AfcdQueue::NewtonStep (void)
{
  NS_LOG_FUNCTION (this);
  uint32_t invsqrt = ((uint32_t) m_recInvSqrt) << REC_INV_SQRT_SHIFT;
  uint32_t invsqrt2 = ((uint64_t) invsqrt * invsqrt) >> 32;
  uint64_t val = (3ll << 32) - ((uint64_t) m_count * invsqrt2);

  val >>= 2; /* avoid overflow */
  val = (val * invsqrt) >> (32 - 2 + 1);
  m_recInvSqrt = val >> REC_INV_SQRT_SHIFT;
}
```

این تابع محاسبه ی مخرج کسر زمان در فورمول محاسبه ی زمان بعدی Drop بسته با توجه به الگوریتم Codel را در اختیار ما قرار می دهد .

```
uint32_t AfcdQueue::ControlLaw (uint32_t t)
{
  NS_LOG_FUNCTION (this);
  return t + ReciprocalDivide (Time2Afcd (m_interval), m_recInvSqrt <<
REC_INV_SQRT_SHIFT);
}
```

این تابع یک ورودی از جنس زمان گرفته و طبق قانون Drop درCodel زمان بعدی حاصل از تقسیم Interval بر حاصل کسر در تابع قبلی را بر می گرداند .

```
uint32_t calculateHash (Ptr<Packet> p){

    //Ptr<Packet> copy = p->Copy ();
    Ipv4Header iph;
    p->RemoveHeader (iph);
    uint32_t SRCIP=iph.GetSource().Get();
    uint32_t DESIP=iph.GetDestination().Get();
    //-----------------------hashing---------------------
    int doing=0;
    bool con=true;
    uint32_t Q=iphashing_key(SRCIP,DESIP);
    uint32_t R=iphashing_step(SRCIP,DESIP);
    while(doing<5 and con)
    {

       if(SHADOW_BUFFER[Q].isfull==0)
       { SHADOW_BUFFER[Q].isfull=1;
         SHADOW_BUFFER[Q].srcip=SRCIP;
         SHADOW_BUFFER[Q].desip=DESIP;
         FLOW_TABLE[Q]=1;
        // std::cout<<"enque ->q:"<<Q<<"----add 1 pack"<<std::endl;
         con=false;
         FlowCount++;
       }
       else if((SHADOW_BUFFER[Q].isfull==1) and
(SHADOW_BUFFER[Q].srcip==SRCIP) and (SHADOW_BUFFER[Q].desip==DESIP) )
       {FLOW_TABLE[Q]++;
         std::cout<<"add to pack"<<std::endl;
        con=false;
        }
        else {
               uint32_t Q=(Q+R)/SHADOW_BUFFER_SIZE;
               doing++;


             }

    }

return Q;
}
```

این تابع یک بسته را به عنوان ورودی گرفته و از بسته آدرس آیپی مبدا و مقصد و پورت مبدا و پورت مقصد را استخراج و آن را Hash **و** Flow Table **و** Shadow Buffer **را آپدیت می کند.**

```
uint32_t fetchhash(Ptr<Packet> p){
    Ptr<Packet> copy = p->Copy ();
    Ipv4Header iph;
    copy->RemoveHeader (iph);
    uint32_t SRCIP=iph.GetSource().Get();
    uint32_t DESIP=iph.GetDestination().Get();
    //---------------------------fetch----------------------
    int doing=0;
    bool con=true;


    uint32_t Q=iphashing_key(SRCIP,DESIP);
    uint32_t R=iphashing_step(SRCIP,DESIP);
    while(doing<5 and con)
    {
       if(SHADOW_BUFFER[Q].isfull==0)
       {
        std::cout<<"no packet1"<<std::endl;
        con=false;
       }
       else if ((SHADOW_BUFFER[Q].isfull==1) and
(SHADOW_BUFFER[Q].srcip==SRCIP) and (SHADOW_BUFFER[Q].desip==DESIP) )
       {

        con=false;
       }
       else
       {
        std::cout<<"no packet2"<<std::endl;
        uint32_t Q=(Q+R)/SHADOW_BUFFER_SIZE;
        doing++;
        Q=-1;
       }


}
     return FLOW_TABLE[Q];
                                    }
```

این تابع یک بسته را به عنوان ورودی گرفته و با محاسبه Hash **مقدار** Flow Table [hash] **را بر میگرداند و** Shadow Buffer **را نیز آپدیت می کند .**

```
Bool AfcdQueue::DoEnqueue (Ptr<Packet> p)
{


  NS_LOG_FUNCTION (this << p);

  if (m_mode == QUEUE_MODE_PACKETS && (m_packets.size () + 1 >
m_maxPackets))
    {
      Drop (p);
      ++m_dropOverLimit;
      return false;
    }

  if (m_mode == QUEUE_MODE_BYTES && (m_bytesInQueue + p->GetSize () >
m_maxBytes))
    {
      Drop (p);
      ++m_dropOverLimit;
      return false;
    }


  Counter++;
  if ((Counter%50)==0){
      qt1=m_packets.size();
        std::cout<<"interval"<<m_interval<<std::endl;
      if (FlowCount!=0)
  mft1=200/FlowCount;
     calculateHash(p);}
  for(int i=0;i<200;i++)
  {if(SHADOW_BUFFER[i].isfull==0){}
      else{
   std::cout<<"i="<<i<<"--"<<FLOW_TABLE[i]<<"--"<<std::endl;}
  }


  AfcdTimestampTag tag;
  p->AddPacketTag (tag);

  m_bytesInQueue += p->GetSize ();
  m_packets.push (p);

  NS_LOG_LOGIC ("Number packets " << m_packets.size ());
```

```
  NS_LOG_LOGIC ("Number bytes " << m_bytesInQueue);

  return true;
}
```

این تابع بسته را به عنوان ورودی گرفته و به ازای هر Sample Packet **تا از بسته های آمده تا به حال از بسته نمونه گرفته و مقدار** Shadow Buffer **و** Flow Table **را به روز می کند . و بسته را در صف قرار می دهد .**

```
bool
AfcdQueue::OkToDrop (Ptr<Packet> p, uint32_t now)
{
  NS_LOG_FUNCTION (this);
  AfcdTimestampTag tag;
  bool okToDrop;
  p->FindFirstMatchingByteTag (tag);
  bool found = p->RemovePacketTag (tag);
  NS_ASSERT_MSG (found, "found a packet without an input timestamp
tag");
  NS_UNUSED (found);    //silence compiler warning
  Time delta = Simulator::Now () - tag.GetTxTime ();
  NS_LOG_INFO ("Sojourn time " << delta.GetSeconds ());
  m_sojourn = delta;
  uint32_t sojournTime = Time2Afcd (delta);
  std::cout<<"delta"<<sojournTime<<std::endl;
  if (FlowCount==0)
  {
    targeti=Time::FromDouble (5000.0, Time::NS);

  }
  else
  {




double mf=mft1+(1)*(qt1-qtarget)-1*(m_packets.size()-qtarget);

    double mi=fetchhash(p);
   std::cout<<"mi"<<mi<<std::endl;
   std::cout<<"mft1"<<mft1<<std::endl;
   std::cout<<"packets"<<m_packets.size()<<std::endl;
if (mf!=0 && mi>0 && mi!=0){

targeti=(targeti*pow(mf,3))/(pow(mi,3));
```

```
}
  }
  if (AfcdTimeBefore (sojournTime, Time2Afcd (targeti))
      || m_bytesInQueue < m_minBytes)
    {
      m_firstAboveTime = 0;
      return false;
    }
  okToDrop = false;
  if (m_firstAboveTime == 0)
    {
      /* just went above from below. If we stay above
       * for at least q->interval we'll say it's ok to drop
       */

      m_firstAboveTime = now + Time2Afcd (m_interval);
    }
  else
  if (AfcdTimeAfter (now, m_firstAboveTime))
    {
      okToDrop = true;
      ++m_state1;
    }


  targeti=Time::FromDouble (5000.0, Time::NS);
  return okToDrop;
}
```

این تابع مدت زمانی که بسته در صف می ماند را محاسبه کرده و با توجه به Shadow و Flow Table **و فورمول های ذکر شده در مقاله حداکثر مدت زمان مجاز ماندن بسته در صف را محاسبه کرده و اگر** Buffer **بسته بیشتر از آن در صف مانده بود مقدار** okTodrop Boolean **را** True **مقدار دهی می کند.**

```
Ptr<Packet>
AfcdQueue::DoDequeue (void)
{


    uint32_t now = AfcdGetTime ();


  NS_LOG_FUNCTION (this);

  if (m_packets.empty ())
    {
```

```
    // وقتی صف خالیست از آن خارج می شویم
    m_dropping = false;
    m_firstAboveTime = 0;
    NS_LOG_LOGIC ("Queue empty");
    return 0;
  }

Ptr<Packet> p = m_packets.front ();
m_packets.pop ();
m_bytesInQueue -= p->GetSize ();




bool okToDrop = OkToDrop (p, now);




if (m_dropping)
  {
     if (!okToDrop)
      {

        m_dropping = false;
      }
    else
    if (AfcdTimeAfterEq (now, m_dropNext))
      {
        m_state2++;
        while (m_dropping && AfcdTimeAfterEq (now, m_dropNext))
          {
            Drop (p);
            ++m_dropCount;
            ++m_count;
            NewtonStep ();
            if (m_packets.empty ())
              {
                m_dropping = false;
                NS_LOG_LOGIC ("Queue empty");
                ++m_states;
                return 0;
              }
            p = m_packets.front ();
```

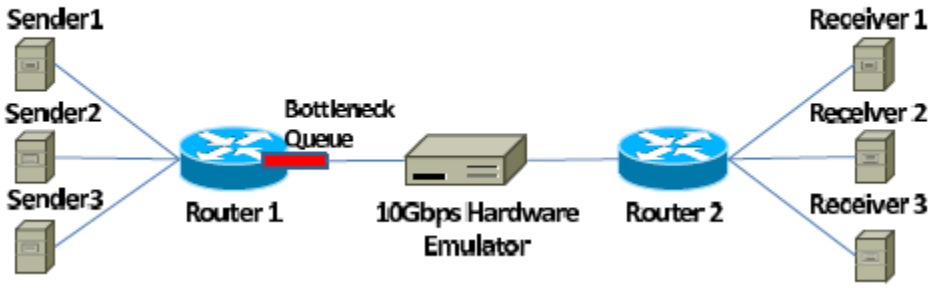
```
              m_packets.pop ();
              m_bytesInQueue -= p->GetSize ();



              if (!OkToDrop (p, now))
                {
                  /* leave dropping state */
                  NS_LOG_LOGIC ("Leaving dropping state");
                  m_dropping = false;
                }
              else
                {
                  /* schedule the next drop */
                  NS_LOG_LOGIC ("Running ControlLaw for input
m_dropNext: " << (double)m_dropNext / 1000000);
                  m_dropNext = ControlLaw (m_dropNext);
                  NS_LOG_LOGIC ("Scheduled next drop at " <<
(double)m_dropNext / 1000000);
                }
            }
        }
    }
  else
    {

      if (okToDrop)
        {
          ++m_dropCount;
          Drop (p);
          if (m_packets.empty ())
            {
              m_dropping = false;
              okToDrop = false;
              NS_LOG_LOGIC ("Queue empty");
              ++m_states;
            }
          else
            {
              p = m_packets.front ();
              m_packets.pop ();
              m_bytesInQueue -= p->GetSize ();

              NS_LOG_LOGIC ("Popped " << p);
              NS_LOG_LOGIC ("Number packets remaining " <<
m_packets.size ());
```

```
              NS_LOG_LOGIC ("Number bytes remaining " <<
m_bytesInQueue);

              okToDrop = OkToDrop (p, now);
              m_dropping = true;
            }
          ++m_state3;

          int delta = m_count - m_lastCount;
          if (delta > 1 && AfcdTimeBefore (now - m_dropNext, 16 *
Time2Afcd (m_interval)))
            {
              m_count = delta;
              NewtonStep ();
            }
          else
            {
              m_count = 1;
              m_recInvSqrt = ~0U >> REC_INV_SQRT_SHIFT;
            }
          m_lastCount = m_count;
          NS_LOG_LOGIC ("Running ControlLaw for input now: " <<
(double)now);
          m_dropNext = ControlLaw (now);
          NS_LOG_LOGIC ("Scheduled next drop at " <<
(double)m_dropNext / 1000000 << " now " << (double)now / 1000000);
        }
    }
  ++m_states;
  return p;
}
```

این تابع بسته ها را در صورت خالی نبودن صف از ابتدای صف برداشته و تابع Oktodrop را برای آن بررسی می کند . سپس بررسی می کند که آیا در فاز Dropping هستیم(در صورتی وارد فاز Dropping می شویم که بسته ای از حداکثر زمان مجاز خود بیشتر در صف باقی بماند) اگر در فاز Dropping بودیم تا زمانی که یا صف خالی شود و یا زمان ماندن بسته ها در صف به زیر مقدار حداکثر مجاز برسد در این فاز با قی می ماند و زمان های Drop بعدی با توجه به فورمول های مو جود در مقاله محاسبه می گردد و اگر در فاز Dropping نبودیم و OktoDrop مقدار True را داشت . وارد فاز Dropping شده و بسته Drop و زمان بعدی Drop محاسبه می گرددو بسته ی بعدی Pop می گردد . و در غیر این صورت صف روال عادی خود را انجام می دهد .

سناریو شبیه سازی :

برای شبیه سازی از سناریو ی شبیه به زیر استفاده شده است .



هر سه Sender **جریان های متمایز و برابر را دارند که هر کدام به** Receiver **متناظر خود داده ارسال می کنند .**

هر سه در لایه ی انتقال از Tcp Reno **استفاده می کنند .**

پهنای باند لینک ها با هم برابر و در حد گیگا بیت می باشد .

سناریو ی بالا در زیر پیاده شده است

```
#include <fstream>
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/flow-monitor-helper.h"
#include "ns3/point-to-point-module.h"
#include "ns3/applications-module.h"
#include "ns3/netanim-module.h"

using namespace ns3;
```

```
NS_LOG_COMPONENT_DEFINE ("RedTests");


std::ofstream myfile ("afcd.txt",std::ofstream::binary);
uint32_t checkTimes;

double avgQueueSize;

// The times


ApplicationContainer sinkApp;

ApplicationContainer sinkApp2;

ApplicationContainer sinkApp3;


NodeContainer n0n2;

NodeContainer n1n2;

NodeContainer n6n2;

NodeContainer n2n3;

NodeContainer n3n4;

NodeContainer n3n5;

NodeContainer n3n7;


Ipv4InterfaceContainer i0i2;

Ipv4InterfaceContainer i1i2;

Ipv4InterfaceContainer i6i2;

Ipv4InterfaceContainer i2i3;

Ipv4InterfaceContainer i3i4;
```

```
Ipv4InterfaceContainer i3i5;

Ipv4InterfaceContainer i3i7;



void CheckQueueSize (Ptr<Queue> queue)

{

//get qsize Red or Droptail or Afcd or Codel

uint32_t qSize = StaticCast<DropTailQueue> (queue)->GetNPackets ();

//uint32_t qSize = StaticCast<CoDelQueue> (queue)->GetNPackets ();

//uint32_t qSize = StaticCast<AfcdQueue> (queue)->GetNPackets ();

//uint32_t qSize = StaticCast<RedQueue> (queue)->GetQueueSize ();



 // Write Queue Size In File P.txt
  //myfile <<qSize;
 // myfile<<"\n";


 uint32_t totalRxBytesCounter = 0;
 uint32_t totalRxBytesCounter2 = 0;
 uint32_t totalRxBytesCounter3 = 0;
 Ptr <Application> app = sinkApp.Get (0);
 Ptr <PacketSink> pktSink = DynamicCast <PacketSink> (app);
 totalRxBytesCounter += pktSink->GetTotalRx ();
 Ptr <Application> app1 = sinkApp2.Get (0);
 Ptr <PacketSink> pktSink1 = DynamicCast <PacketSink> (app1);
 totalRxBytesCounter2 += pktSink1->GetTotalRx ();
 Ptr <Application> app2 = sinkApp3.Get (0);
 Ptr <PacketSink> pktSink2 = DynamicCast <PacketSink> (app2);
 totalRxBytesCounter3 += pktSink2->GetTotalRx ();

// totalRxBytesCounter3 += pktSink2->GetTotalRx ();
//  "trouph flow 1 to 1";
//myfile <<totalRxBytesCounter/Simulator::Now ().GetSeconds ();
//myfile<<"\n";
// trouph flow 2 to 2;
```

```
//  myfile <<totalRxBytesCounter2/Simulator::Now ().GetSeconds ();
// myfile<<"\n";
// trouph  flow 3 to 3;
// myfile <<totalRxBytesCounter3/Simulator::Now ().GetSeconds ();

// "trouph all";
//  myfile
<<(totalRxBytesCounter2+totalRxBytesCounter+totalRxBytesCounter3)/Simu
lator::Now ().GetSeconds ();
//  myfile<<"\n";

 Simulator::Schedule (Seconds (0.1), &CheckQueueSize, queue);




}



int

main (int argc, char *argv[])

{

//  std::string animFile = "dumbbell3-animation.xml" ; // Name of file
for animation output


  LogComponentEnable ("RedQueue", LOG_LEVEL_INFO);



  std::string redLinkDataRate = "1000Mbps";

  std::string redLinkDelay = "120ms";
```

```
  NS_LOG_INFO ("Create nodes");

  NodeContainer c;

  c.Create (8);

  Names::Add ( "N0", c.Get (0));

  Names::Add ( "N1", c.Get (1));

  Names::Add ( "N2", c.Get (2));

  Names::Add ( "N3", c.Get (3));

  Names::Add ( "N4", c.Get (4));

  Names::Add ( "N5", c.Get (5));

  Names::Add ( "N6", c.Get (6));

  Names::Add ( "N7", c.Get (7));


  n0n2 = NodeContainer (c.Get (0), c.Get (2));

  n1n2 = NodeContainer (c.Get (1), c.Get (2));

  n6n2=NodeContainer (c.Get (6), c.Get (2));

  n2n3 = NodeContainer (c.Get (2), c.Get (3));

  n3n4 = NodeContainer (c.Get (3), c.Get (4));

  n3n5 = NodeContainer (c.Get (3), c.Get (5));

  n3n7=NodeContainer (c.Get (3), c.Get (7));

  Config::SetDefault ("ns3::TcpL4Protocol::SocketType", StringValue
("ns3::TcpReno"));

 // 42 = headers size

 //  Config::SetDefault ("ns3::TcpSocket::SegmentSize", UintegerValue
(1000 - 42));
```

```
 // GlobalValue::Bind ("ChecksumEnabled", BooleanValue (false));

 Config::SetDefault ("ns3::DropTailQueue::MaxPackets", UintegerValue
(50));

 //uint32_t meanPktSize = 500;









  NS_LOG_INFO ("Install internet stack on all nodes.");

  InternetStackHelper internet;

  internet.Install (c);


 NS_LOG_INFO ("Create channels");

  PointToPointHelper p2p;


  p2p.SetQueue ("ns3::DropTailQueue","Mode",StringValue
("QUEUE_MODE_BYTES"));

  p2p.SetDeviceAttribute ("DataRate", StringValue ("4000Mbps"));

  p2p.SetChannelAttribute ("Delay", StringValue ("1ms"));

  NetDeviceContainer devn0n2 = p2p.Install (n0n2);

  NetDeviceContainer devn6n2 = p2p.Install (n6n2);

  p2p.SetQueue ("ns3::DropTailQueue");

  p2p.SetDeviceAttribute ("DataRate", StringValue ("4000Mbps"));

  p2p.SetChannelAttribute ("Delay", StringValue ("1ms"));
```

```
  NetDeviceContainer devn1n2 = p2p.Install (n1n2);


 //Config::SetDefault ("ns3::CodelQueue::Mode", StringValue
("QUEUE_MODE_PACKETS"));

 // Config::SetDefault ("ns3::CodelQueue::MaxPackets", UintegerValue
(1000));

 // Config::SetDefault ("ns3::RedQueue::QueueLimit", UintegerValue
(maxPackets));

 //Config::SetDefault ("ns3::RedQueue::Mode", StringValue
("QUEUE_MODE_PACKETS"));

 //Config::SetDefault ("ns3::RedQueue::MeanPktSize", UintegerValue
(meanPktSize));

 // Config::SetDefault ("ns3::RedQueue::Wait", BooleanValue (true));

 // Config::SetDefault ("ns3::RedQueue::Gentle", BooleanValue (true));

 // Config::SetDefault ("ns3::RedQueue::QW", DoubleValue (0.002));

 Config::SetDefault ("ns3::RedQueue::MinTh", DoubleValue (25));

 Config::SetDefault ("ns3::RedQueue::MaxTh", DoubleValue (40));

 Config::SetDefault ("ns3::RedQueue::QueueLimit", UintegerValue (50));

 p2p.SetQueue ("ns3::DropTailQueue");

 //  "LinkDelay", StringValue (redLinkDelay));


  p2p.SetDeviceAttribute ("DataRate", StringValue (redLinkDataRate));
  p2p.SetChannelAttribute ("Delay", StringValue (redLinkDelay));

  NetDeviceContainer devn2n3 = p2p.Install (n2n3);




  p2p.SetQueue ("ns3::DropTailQueue");

  p2p.SetDeviceAttribute ("DataRate", StringValue ("4000Mbps"));
```

```
p2p.SetChannelAttribute ("Delay", StringValue ("1ms"));
NetDeviceContainer devn3n4 = p2p.Install (n3n4);
NetDeviceContainer devn3n7= p2p.Install (n3n7);
p2p.SetQueue ("ns3::DropTailQueue");
p2p.SetDeviceAttribute ("DataRate", StringValue ("4000Mbps"));
p2p.SetChannelAttribute ("Delay", StringValue ("1ms"));
NetDeviceContainer devn3n5 = p2p.Install (n3n5);


NS_LOG_INFO ("Assign IP Addresses");
Ipv4AddressHelper ipv4;

ipv4.SetBase ("10.1.1.0", "255.255.255.0");
i0i2 = ipv4.Assign (devn0n2);

ipv4.SetBase ("10.1.2.0", "255.255.255.0");
i1i2 = ipv4.Assign (devn1n2);

ipv4.SetBase ("10.1.3.0", "255.255.255.0");
i2i3 = ipv4.Assign (devn2n3);

ipv4.SetBase ("10.1.4.0", "255.255.255.0");
i3i4 = ipv4.Assign (devn3n4);
```

```
  ipv4.SetBase ("10.1.5.0", "255.255.255.0");

  i3i5 = ipv4.Assign (devn3n5);

  ipv4.SetBase ("10.1.6.0", "255.255.255.0");

  i6i2 = ipv4.Assign (devn6n2);

  ipv4.SetBase ("10.1.7.0", "255.255.255.0");

  i3i7 = ipv4.Assign (devn3n7);



  // Set up the routing

  Ipv4GlobalRoutingHelper::PopulateRoutingTables ();

  // SINK is in the right side

  uint16_t port = 50000;

  Address sinkLocalAddress (InetSocketAddress (Ipv4Address::GetAny (),
port));

  PacketSinkHelper sinkHelper ("ns3::TcpSocketFactory",
sinkLocalAddress);



 sinkApp = sinkHelper.Install (n3n4.Get (1));
  \
 sinkApp2 = sinkHelper.Install (n3n5.Get (1));

 sinkApp3 = sinkHelper.Install (n3n7.Get (1));

 sinkApp.Start (Seconds (0));

 sinkApp.Stop (Seconds (11));

 sinkApp2.Start (Seconds (0));

 sinkApp2.Stop (Seconds (11));
```

```
 sinkApp3.Start (Seconds (0));
 sinkApp3.Stop (Seconds (11));


  // Connection one
  // Clients are in left side
  /*
   * Create the OnOff applications to send TCP to the server
   * onoffhelper is a client that send data to TCP destination
   */
  OnOffHelper clientHelper1 ("ns3::TcpSocketFactory", Address ());
  clientHelper1.SetAttribute
    ("OnTime", StringValue
("ns3::ConstantRandomVariable[Constant=1]"));
  clientHelper1.SetAttribute
    ("OffTime", StringValue
("ns3::ConstantRandomVariable[Constant=0]"));
  clientHelper1.SetAttribute
    ("DataRate", DataRateValue (DataRate ("4000Mb/s")));
 clientHelper1.SetAttribute
    ("PacketSize", UintegerValue (1000));


  ApplicationContainer clientApps1;
  AddressValue remoteAddress
    (InetSocketAddress (i3i4.GetAddress (1), port));
```

```
  clientHelper1.SetAttribute ("Remote", remoteAddress);
  AddressValue remoteAddress2

    (InetSocketAddress (i3i5.GetAddress (1), port));

  AddressValue remoteAddress3

    (InetSocketAddress (i3i7.GetAddress (1), port));

  clientHelper1.SetAttribute ("Remote", remoteAddress);

  clientApps1.Add (clientHelper1.Install (n0n2.Get (0)));

  clientApps1.Start (Seconds (0));

  clientApps1.Stop (Seconds (11));


 // Connection two

  OnOffHelper clientHelper2 ("ns3::TcpSocketFactory", Address ());

  clientHelper2.SetAttribute

    ("OnTime", StringValue
("ns3::ConstantRandomVariable[Constant=1]"));

  clientHelper2.SetAttribute

    ("OffTime", StringValue
("ns3::ConstantRandomVariable[Constant=0]"));

  clientHelper2.SetAttribute

   ("DataRate", DataRateValue (DataRate ("4000Mb/s")));

  clientHelper2.SetAttribute

    ("PacketSize", UintegerValue (1000));


  ApplicationContainer clientApps2;
```

```
  clientHelper2.SetAttribute ("Remote", remoteAddress2);
  clientApps2.Add (clientHelper2.Install (n1n2.Get (0)));
  clientApps2.Start (Seconds (0.0));
  clientApps2.Stop (Seconds (11));



  // Connection 3
   OnOffHelper clientHelper3 ("ns3::TcpSocketFactory", Address ());
   clientHelper3.SetAttribute
     ("OnTime", StringValue
("ns3::ConstantRandomVariable[Constant=1]"));
   clientHelper3.SetAttribute
     ("OffTime", StringValue
("ns3::ConstantRandomVariable[Constant=0]"));
   clientHelper3.SetAttribute
    ("DataRate", DataRateValue (DataRate ("4000Mb/s")));
   clientHelper3.SetAttribute
     ("PacketSize", UintegerValue (1000));


   ApplicationContainer clientApps3;
   clientHelper3.SetAttribute ("Remote", remoteAddress3);
   clientApps3.Add (clientHelper3.Install (n6n2.Get (0)));
   clientApps3.Start (Seconds (0.0));
   clientApps3.Stop (Seconds (11));
```

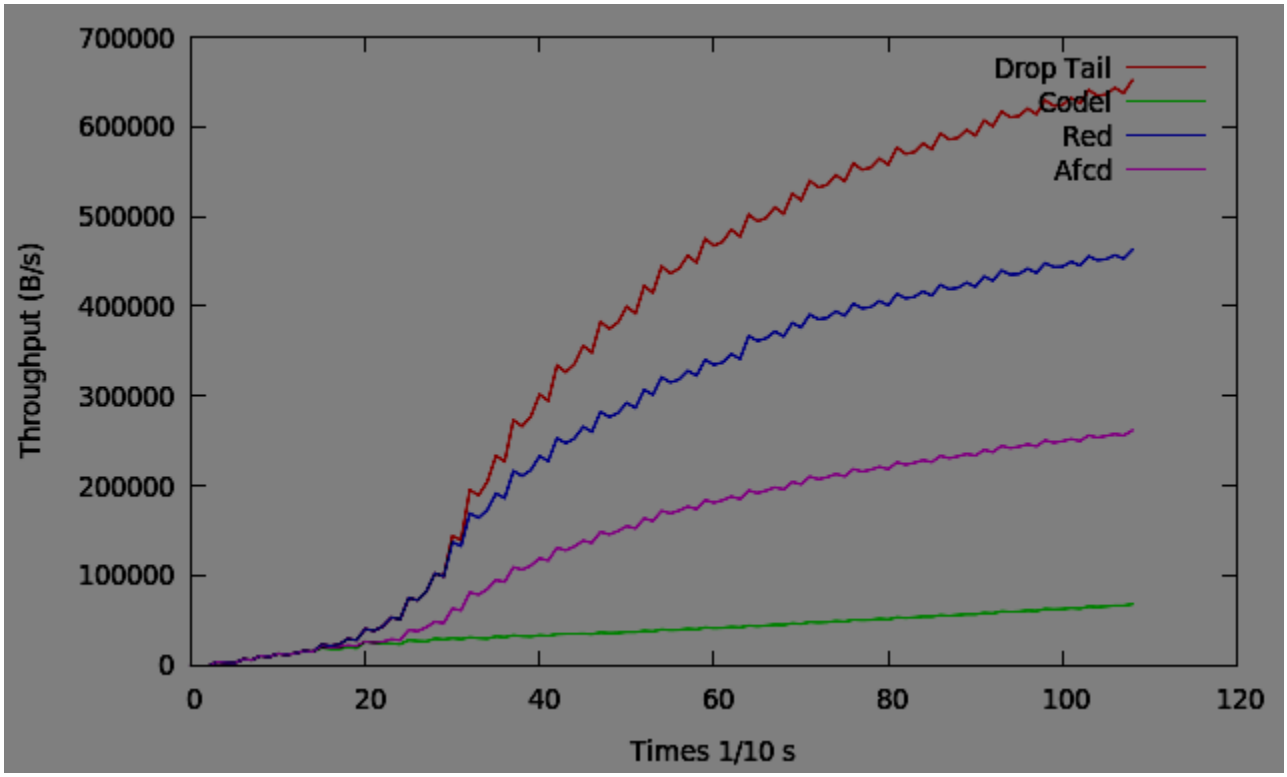
```
    Ptr<PointToPointNetDevice> nd = StaticCast<PointToPointNetDevice>
(devn2n3.Get (0));

    Ptr<Queue> queue = nd->GetQueue ();

   Simulator::ScheduleNow (&CheckQueueSize, queue);



   // Create the animation object and configure for specified output
   //AnimationInterface anim (animFile);
   //anim.EnablePacketMetadata (); // Optional
   //anim.EnableIpv4L3ProtocolCounters (Seconds (0), Seconds (100));
// Optional



  Simulator::Stop (Seconds (11));

  Simulator::Run ();




Simulator::Destroy ();


 return 0;

}
```

نمودار ها :

مقایسه ی کارایی :

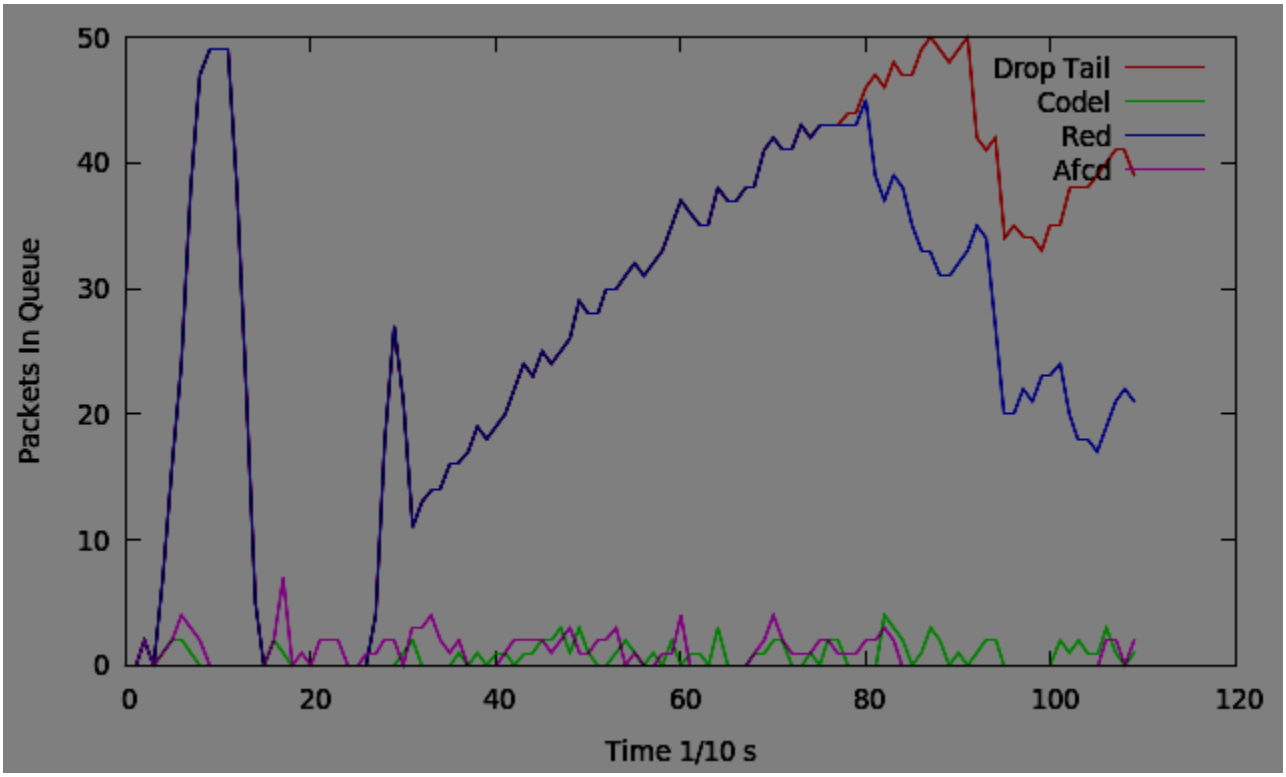
تمام شبیه سازی ها در 12 **ثانیه ابتدایی انجام شده است .**

با اجرای سناریو به ازای چهار مدل صف Afcd-odel-DropTail-Red **و ثبت کردن کارایی در** event **های** 1 **ثانیه ای نمودار زیر به دست می آید .**



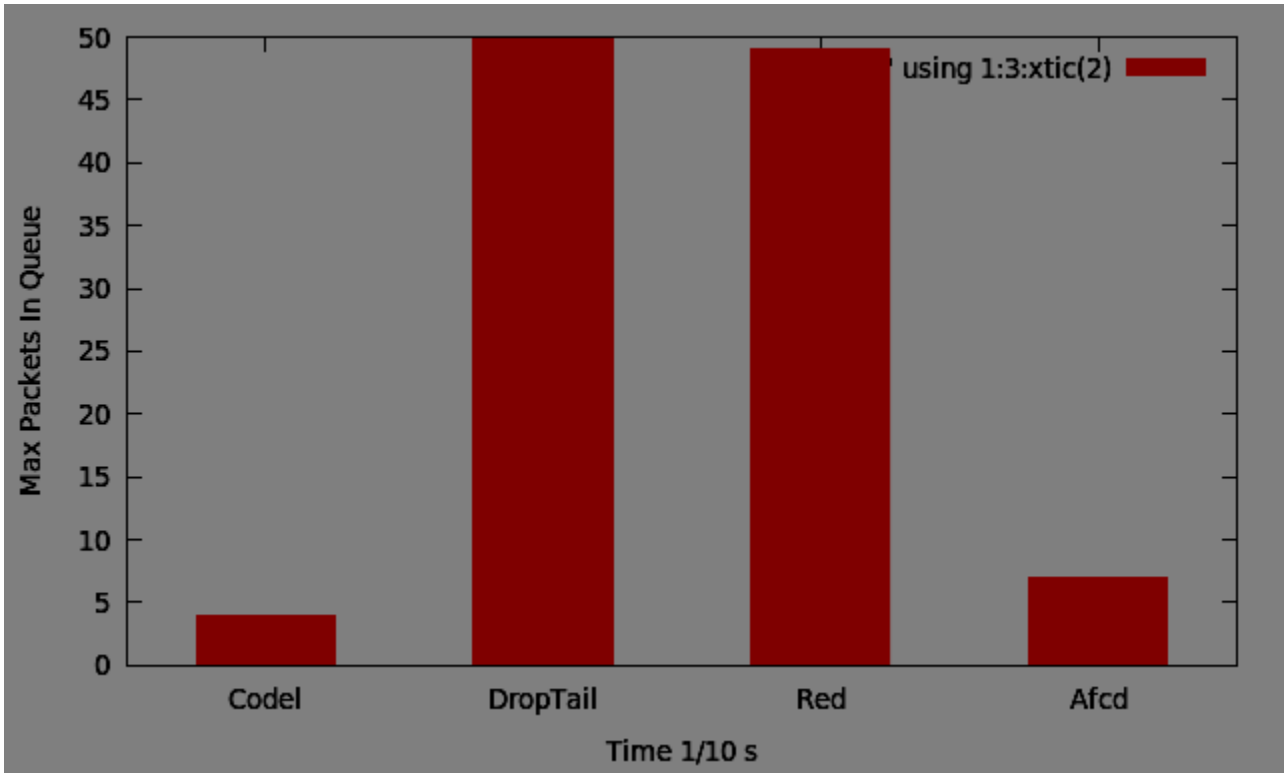
مقایسه ی تعداد بسته ها در صف :

با اجرای سناریو به ازای چهار مدل صف ذکر شده و ثبت کردن طول صف در Event **های** 1 **ثانیه ای نمودار زیر به دست می آید . همانطور که مشاهده می کنید** Drop Tail **و** Red **تعداد بسته های زیادی در صف دارند .اما** Codel **و** Afcd **تقریبا طول صف پایین و نوسان آنها نیز نسبت به** Red **و** Codel **نیز پایین است .**



حداکثر تعداد بسته ها در صف

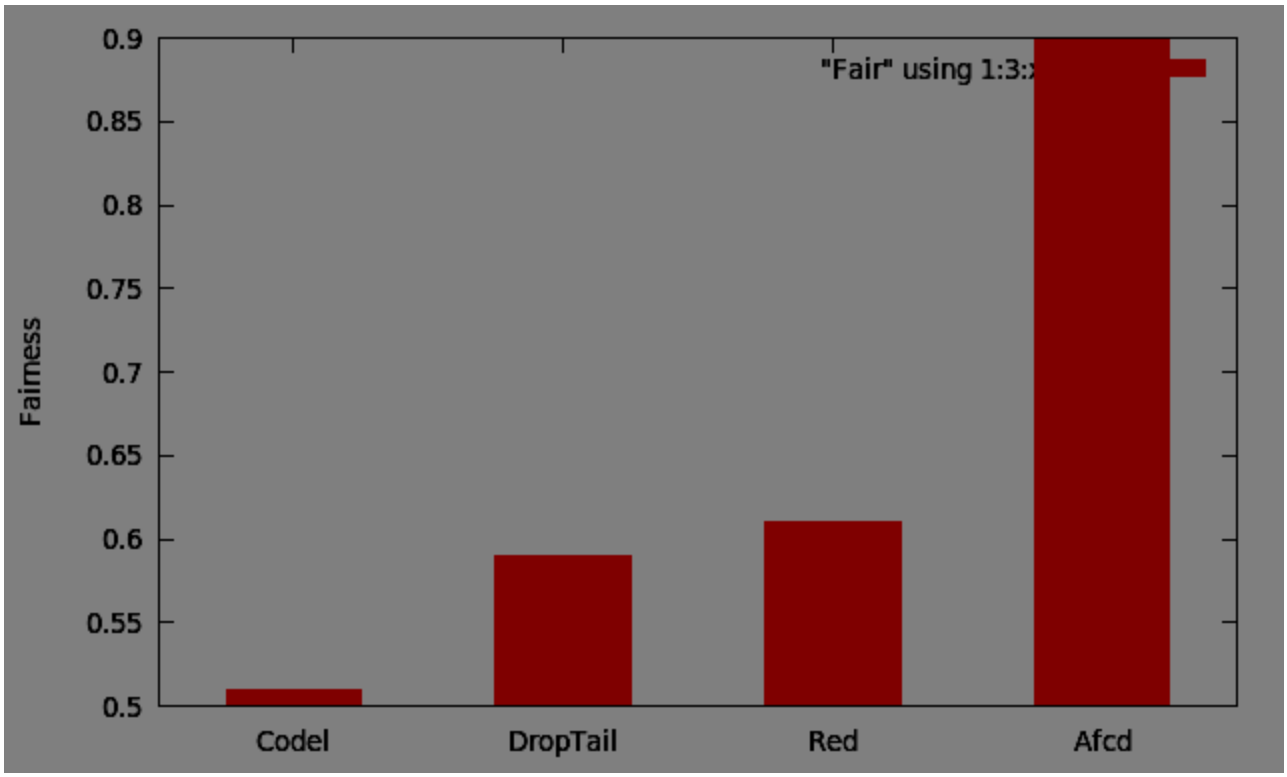
طول بافر صف در هر 4 **مکانیسم** 50 **بسته در نظر گرفته شده همانطور که مشاهده می کنید .** Drop Tail **و** Red **به ازدحام و پر شدن حداکثر صف رسیدن اما** Codel **و** Afcd **صف را پر نکرده اند . حتی به نیمی از طول صف پر هم نرسیده اند .**



نمودار Fairness

نمودار عدالت نشان دهنده ی متوسط نسبت کارایی تمام جریان ها به صورت دو به دو می باشد .

برای رسم این نمودار ابتدا نسبت کارایی هر دو جریان نسبت به هم به ازای 110 **بار برای هر کدام را به دست آوردیم و سپس متوسط این نسبت را به دست آوردیم که حاصل در نمودار زیر نشان داده شده است .**


"Fair" using 1:3:
Faimess
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
Codel
DropTail
Red
Afcd