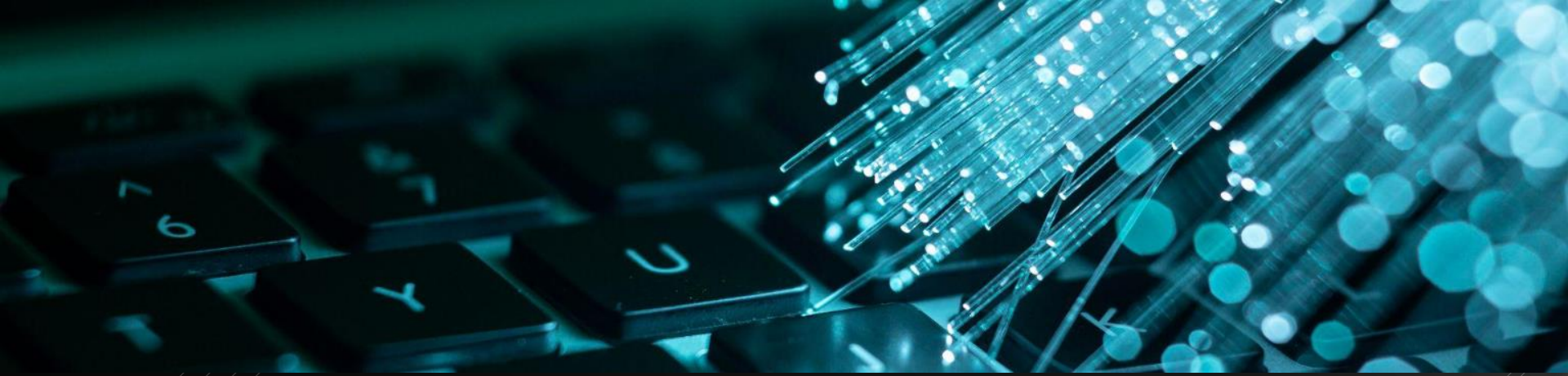
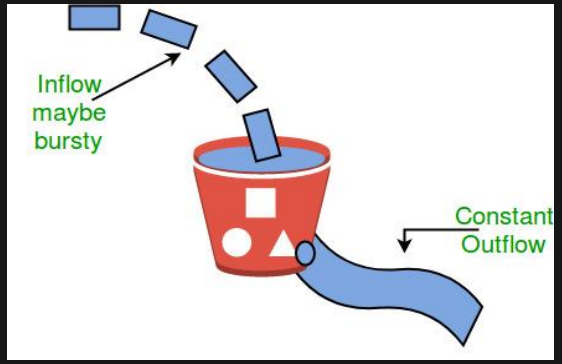


Data Center TCP
DCTCP

Socea Mihai, Horduna Emilian, Giuglan Catalin, Corban Alexandru

Controlul congestiei



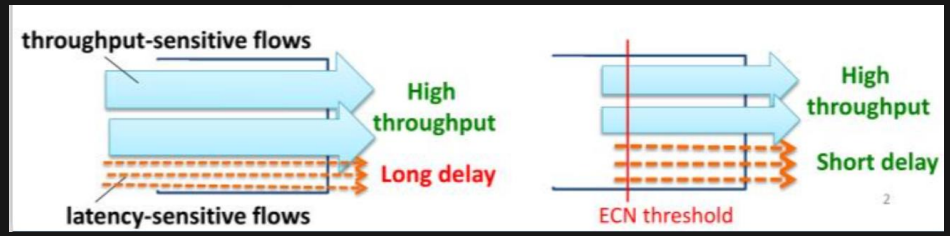
Definitie

Controlul congestionării reprezintă un set de mecanisme și tehnici utilizate în rețelele de calculatoare pentru a preveni și a gestiona suprasolicitarea rețelei, asigurând astfel un flux eficient și stabil de date.

Obiective

- Prevenirea pierderilor de pachete
- Minimizarea latenței
- Optimizarea utilizării benzii de rețea
- Stabilitatea rețelei
- Corectitudinea alocării resurselor
- Scalabilitate
- Adaptabilitate
- Evitarea supraîncărcării

DCTCP



Definitie

Este un protocol de transport optimizat pentru rețelele de centre de date, dezvoltat pentru a îmbunătăți performanța în medii cu latente scăzute și rate mari de transfer de date. Acesta este o variantă imbunatatita a protocolului TCP, utilizand mecanisme specifice (ex: feedback-ul explicit de la echipamentele de rețea)

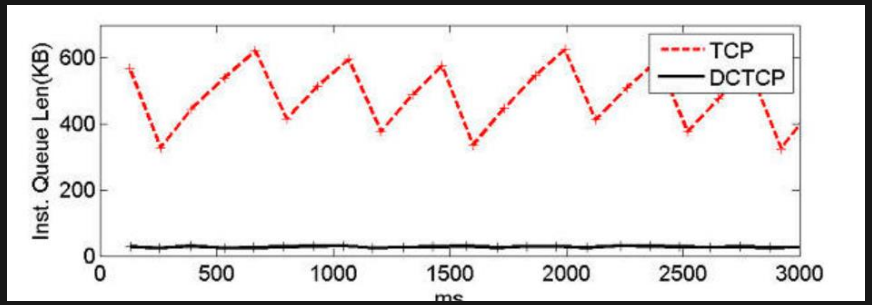
Caracteristici

- Feedback explicit de congestie (ECN) : primește semnale de avertizare din timp, permițând ajustări prompte ale ratei de transmisie
- Reglarea precisă a ratei de transmisie
- Reducerea latenței
- Eficiență crescută a utilizării bandwidth (mai putine pierderi de pachete)

DCTCP VS TCP

Motivatie

În centrele de date, TCP-ul tradițional nu reușește să gestioneze eficient congestia rețelei, ceea ce duce la formarea cozilor în switch-uri și la întârzieri fluctuante. Pentru a reduce aceste probleme, Windows Server 2012 introduce DCTCP, care folosește Explicit Congestion Notification (ECN) pentru a estima nivelul de congestie și a ajusta rata de trimitere a pachetelor corespunzător.



Demonstratie vizuala

Ilustrația următoare demonstrează eficiența DCTCP în a atinge un throughput complet, ocupând în același timp un spațiu foarte mic în buffer-ul de pachete al unui switch Ethernet, comparativ cu TCP-ul tradițional.

Algoritmul DCTCP

Estimare Alpha

$$\alpha = \alpha \times (1 - g) + g \times F$$

α = estimare (facuta de sender) a fracției de pachete marcate, numită α, care este actualizată o singură dată pentru fiecare fereastră de date
g = factorul de ajustare a estimării, un număr real între 0 și 1
F = fracția de octeți trimiși care au întâlnit congestionare în timpul ferestrei de observație anterioare.

$$F = \frac{\text{Octeți marcați cu ECN}}{\text{Totalul octeților trimiși în timpul ferestrei de observație}}$$

**ECN (Explicit Congestion Notification) este o funcționalitate în rețelele de calculatoare, folosită pentru a semnala și gestiona congestionarea fără a pierde pachete.

$$\text{CWND} \leftarrow \text{CWND} \times \left(1 - \frac{\alpha}{2}\right)$$

Ajustarea ferestrei de congestie (CWND)

Sender: Conectarea sursei la receptor

```
void CCSrc::connect(Route* routeout, Route* routeback, CCSink& sink, simtime_picosec starttime) {
    assert(routeout);  // Verifică dacă ruta de ieșire este validă
    _route = routeout; // Stabilește ruta de ieșire
    _sink = &sink;     // Asociază receptorul (sink) cu sursa
    _flow._name = _name; // Setează numele fluxului
    _sink->connect(*this, routeback); // Conectează receptorul la sursă folosind ruta de întoarcere

    eventlist().sourceIsPending(*this, starttime); // Planifică evenimentul de început al transmisiei
}
```

S: Procesare Nack-uri

```
void CCSrc::processNack(const CCNack& nack){
    _nacks_received++;  // Incrementează contorul de NACK-uri primite
    _flightsize -= _mss; // Reduce dimensiunea ferestrei de zbor cu dimensiunea segmentului

    if (nack.ackno() >= _next_decision) {
        _cwnd = _cwnd / 2;  // Reduce fereastra de congestie la jumătate
        if (_cwnd < _mss)
            _cwnd = _mss;  // Asigură că fereastra de congestie nu scade sub dimensiunea minimă a segmentului

        _ssthresh = _cwnd;  // Actualizează pragul de start lent
        _next_decision = _highest_sent + _cwnd;  // Setează următorul punct de decizie pentru controlul congestiei
    }
}
```

**NACK (Negative Acknowledgment): Un semnal care indică faptul că un pachet nu a fost primit corect și trebuie retransmis.

S: Procesare Ack-uri

```
void CCSrc::processAck(const CCAck& ack) {
    CCAck::seq_t ackno = ack.ackno(); // Obține numărul de secvență al ACK-ului
    _acks_received++;  // Incrementează contorul de ACK-uri primite
    _flightsize -= _mss; // Reduce dimensiunea ferestrei de zbor cu dimensiunea segmentului

    uint64_t bytes_acked = ackno - _highest_sent; // Calculează numărul de octeți confirmați
    _bytes_acked += bytes_acked; // Actualizează numărul total de octeți confirmați

    if (ack.is_ecn_marked()){ // Verifică dacă pachetul a fost marcat cu ECN
        _bytes_marked += bytes_acked; // Actualizează numărul de octeți marcați
    }

    // Calculează fracțiunea de octeți marcați
    double M = (double)_bytes_marked / _bytes_acked;

    // Actualizează valoarea _alpha
    _alpha = _alpha * (1 - _g) + _g * M;

    // Ajustează fereastra de congestie pe baza valorii _alpha
    _cwnd = _cwnd * (1 - _alpha / 2);
    if (_cwnd < _mss)
        _cwnd = _mss;

    _next_decision = _highest_sent + _cwnd;

    // Resetează contoarele de octeți
    _bytes_acked = 0;
    _bytes_marked = 0;

    // Ajustează rata de trimitere
    if (_cwnd < _ssthresh)
        _cwnd += _mss;  // Slow start
    else
        _cwnd += _mss * _mss / _cwnd;  // Congestion avoidance
}
```

**ACK (Acknowledgment): Un semnal care confirmă că un pachet a fost primit corect.

S: Receive & Send packet

```
void CCSrc::receivePacket(Packet& pkt)
{
    if (!_flow_started){
        return;
    }

    switch (pkt.type()) {
    case CCNACK:
        processNack((const CCNack&)pkt);
        pkt.free();
        break;
    case CCACK:
        processAck((const CCAck&)pkt);
        pkt.free();
        break;
    default:
        cout << "Got packet with type " << pkt.type() << endl;
        abort();
    }

    //now send packets!
    while (_flightsize + _mss < _cwnd)
        send_packet();
}
```

R: conexiunea rec -> sursa

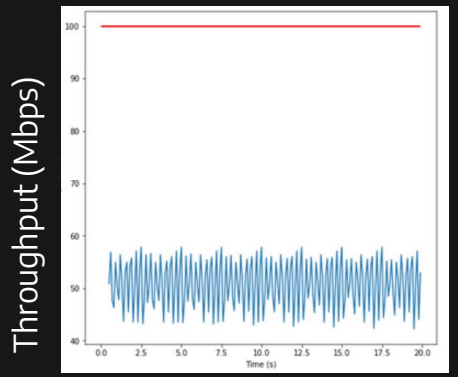
```
void CCSink::connect(CCSrc& src, Route* route)
{
    _src = &src;
    _route = route;
    setName(_src->_nodename);
}
```

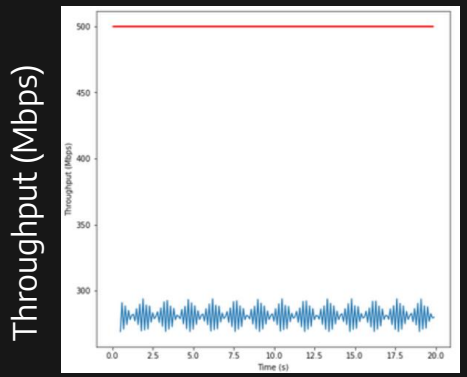
```
void CCSrc::send_packet() {
    CCPacket* p = NULL;

    assert(_flow_started);

    p = CCPacket::newpkt(*_route,_flow, _highest_sent+1, _mss, eventlist().now());

    _highest_sent += _mss;
    _packets_sent++;

    _flightsize += _mss;

    p->sendOn();
}
```

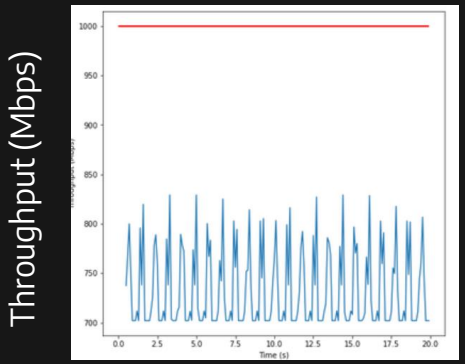
Receiver: Gestionarea pachetelor

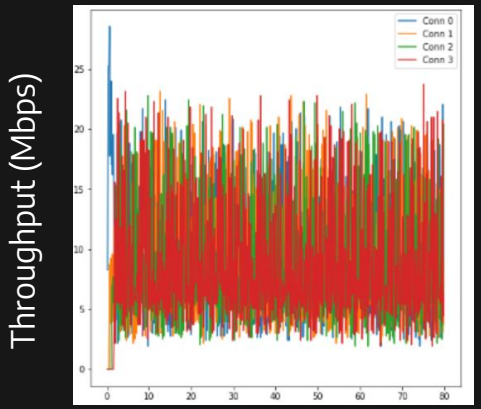
```
void CCSink::receivePacket(Packet& pkt) {
    switch (pkt.type()) {
    case CC:
        break;
    default:
        abort();
    }

    CCPacket *p = (CCPacket*)(&pkt);
    CCPacket::seq_t seqno = p->seqno();

    simtime_picosec ts = p->ts();

    if (pkt.header_only()){
        send_nack(ts,seqno);

        p->free();
        return;
    }

    int size = p->size()-ACKSIZE;
    _total_received += Packet::data_packet_size();;

    bool ecn = (bool)(pkt.flags() & ECN_CE);

    send_ack(ts,seqno,ecn);
    pkt.free();
}

void CCSink::send_ack(simtime_picosec ts,CCPacket::seq_t ackno,bool ecn)
void CCSink::send_nack(simtime_picosec ts, CCPacket::seq_t ackno)
```

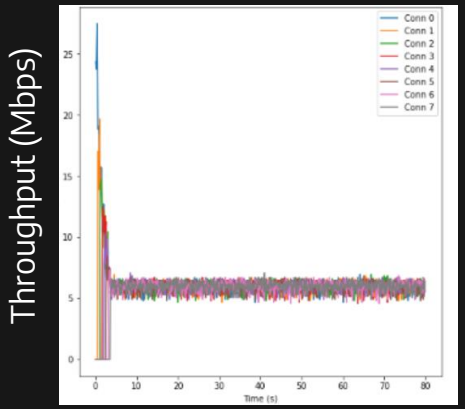
Rezultate grafice

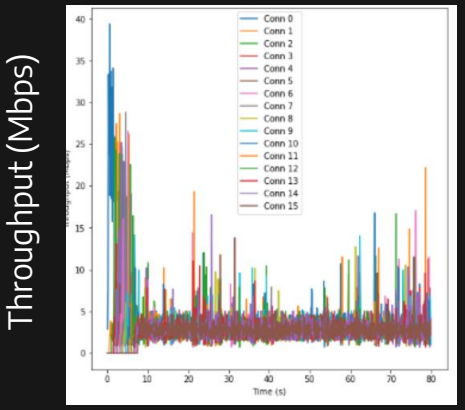


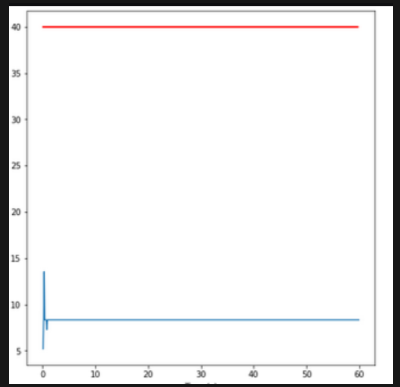


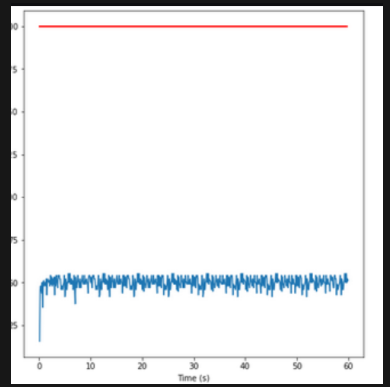


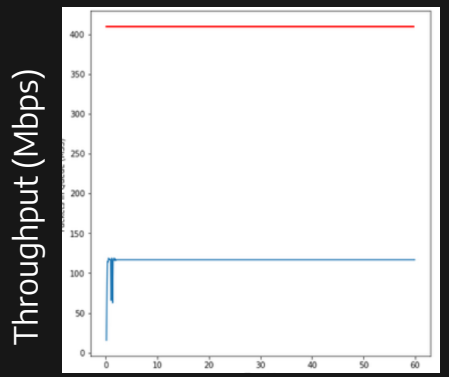

Throughput (Mbps)
Conn 0
Conn 1
Conn 2
Conn 3
Time (s)


Throughput (Mbps)
Conn 0
Conn 1
Conn 2
Conn 3
Conn 4
Conn 5
Conn 6
Conn 7
Time (s)


Throughput (Mbps)
Conn 0
Conn 1
Conn 2
Conn 3
Conn 4
Conn 5
Conn 6
Conn 7
Conn 8
Conn 9
Conn 10
Conn 11
Conn 12
Conn 13
Conn 14
Conn 15
Time (s)


Throughput (Mbps)


Throughput (Mbps)
Time (s)


Throughput (Mbps)

Mulțumim pentru atenție!