Poseidon: Efficient, Robust, and Practical Datacenter CC via Deployable INT

Weitao Wang, Masoud Moshref, Yuliang Li, Gautam Kumar,
T. S. Eugene Ng, Neal Cardwell, Nandita Dukkipati
A Good Congestion Control Algorithm

Design Principles | Motivation | Key Idea | Design | Evaluation
--- | --- | --- | --- | ---
Max-min Fair | Efficient | High utilization, Low latency, Fast convergence, Stable rate after convergence | Practical | Low overhead, Incremental deployment
Motivation 1: React to Every Congestion -> Not Max-min Fairness

The fair-share for the victim flow changes when new flows join.
Motivation 1: React to Every Congestion -> Not Max-min Fairness

Victim flow that travels 2 saturated links

\[ M = 2 \]
\[ N = 2 \]

Line rate: 200 Gbps

The fair-share for the victim flow changes when new flows join.
Motivation 1: React to Every Congestion -> Not Max-min Fairness

Victim flow that travels 2 saturated links

Line rate: 200 Gbps

The fair-share for the victim flow changes when new flows join.
Motivation 1: React to Every Congestion -> Not Max-min Fairness

The fair-share for the victim flow changes when new flows join.
Motivation 1: React to Every Congestion -> Not Max-min Fairness

Swift reacts to end-to-end fabric delay, so the victim flow has a much higher fabric delay.
Motivation 1: React to Every Congestion -> Not Max-min Fairness

HPCC & DCTCP react to every congestion, so the victim flow does more MD operations.
Motivation 2: Decrease rate below fair-share -> slow convergence

The flow that haven’t reached fair-share should not decrease rate.
Motivation 3: Convergence Speed & Stable Rate Trade-off

AIMD uses fixed AI step, so it cannot achieve both fast convergence and stable rate enforcement.
Not max-min Fairness  \rightarrow \text{Decrease before fair-share}  \rightarrow \text{Convergence & stability trade-off}

React to every congestion  \rightarrow \text{AIMD demands same reaction from all flows}  \rightarrow \text{Binary signal}

React to bottleneck congestion  \rightarrow \text{Quantitative signal}

In-network Telemetry (INT) \rightarrow \text{Enable}
Design 1: A Practical Low-overhead Quantitative Signal

- Signal: maximum per-hop delay (MPD)
  - Fixed short length: 2 bytes
  - Collected along the forwarding path
  - Reflected to sender through ACK

Queue delay:
- 3 us
- 20 us
- 15 us
Why does **Existing CC with INT** Have the Same Problems?

They still uses same idea as AIMD

Either all flows increase, or all flows decrease

Poseidon decouples from AIMD

Every flow **reacts differently,**
Some increase, some decrease.
Design 2: Rate-adaptive Target Enables Different Reactions

- Each flow calculates its own max per-hop delay target (MPT)
  - \( MPT = T(rate) \)
  - larger rate \( \rightarrow \) smaller target
Design 3: Adaptive MIMD Rate Update

• Each flow updates rate multiplicatively (MIMD)
  • update_ratio = U(MPT, MPD)
  • new_rate = rate * update_ratio

• MPT < MPD, decrease
  • MPT << MPD, decrease more drastic

• MPT > MPD, increase
  • MPT >> MPD, increase more drastic
Convergence to Single-hop Fairness

Flow A rate: $a$
Flow B rate: $b$  (assume $a < b$)
**Convergence to Single-hop Fairness**

Flow A rate: $a$

Flow B rate: $b$  (assume $a < b$)

Goal: update the rates to be in “more fair” area.
Convergence to Single-hop Fairness

Flow A rate: a
Flow B rate: b  (assume a < b)
Goal: update the rates to be in “more fair” area.

Given any delay D, the rate updates are:

\[ a' = a \cdot U(T(a), D) \]
\[ b' = b \cdot U(T(b), D) \]

To guarantee convergence:

\[ \frac{a}{b} < \frac{b'}{a'} < \frac{b}{a} \]
Convergence to Single-hop Fairness

Flow A rate: \(a\)
Flow B rate: \(b\)  (assume \(a < b\))

Goal: update the rates to be in “more fair” area.

Given any delay \(D\), the rate updates are:

\[
\begin{align*}
    a' &= a \cdot U(T(a), D) \\
    b' &= b \cdot U(T(b), D)
\end{align*}
\]

To guarantee convergence:

\[
\frac{a}{b} < \frac{b'}{a'} < \frac{b}{a}
\]

Repeat until converge.

Note: The complete proof with corner cases discussion is in the paper.
Convergence to Single-hop Fairness

Flow A rate: $a$
Flow B rate: $b$ (assume $a < b$)

Goal: update the rates to be in "more fair" area.

Given any delay $D$, the rate updates are:

$$a' = a \cdot U(T(a), D)$$
$$b' = b \cdot U(T(b), D)$$

To guarantee convergence:

$$\frac{a}{b} < \frac{b'}{a'} < \frac{b}{a}$$

Repeat until converge.

Any update function $U()$ and target function $T()$ need to satisfy this inequality.

Note: The complete proof with corner cases discussion is in the paper.
Convergence to Max-min Fairness in a Network

Red flow’s MPD = \( \max(D1, D2) = D1 \)

The bottleneck always has the largest delay. We proved this leads to max-min fairness.
Implementation

• Testbed
  • Implementation
    • 2 lines of core P4 code to obtain INT signal
    • Small changes to Swift algorithm in Pony Express
  • Topology
    • 2 hosts (virtualized into 16 hosts) + 2 Tofino-2 switches

• Simulator
  • Customized OMNeT++ packet simulator
  • Topology
    • Clos network with 64 racks
Evaluation Summary

- **Efficiency**
  - **12x** faster convergence
  - **24x** more stable throughput
  - **3x** lower RTT
  - **Full** utilization
  - **1.78x** faster median and **27x** faster tail op latency (FCT)

- **Robustness - max-min fairness**
  - Max-min fair in **multi-hop** congestion
  - Max-min fair in **reverse-path** congestion

- **Practical**
  - Implementation on production networking stack with no NIC changes
  - **Incremental gain** for incremental deployment
  - Bounded unfairness during partial deployment
Fast Convergence and Stable Throughput

**12x Faster Convergence**
Faster multiplicative increase.
Ramp-up without any decrease.

**24x More Stable Throughput**
Do not need additive increase.
Update $U() = 1.0$ after converge.
Poseidon Achieves Max-min Fairness

Poseidon achieves max-min fair rate for all flows, including the victim flow.

Line rate: 200 Gbps
### Performance Gain for Incremental Deployment

4 racks send traffic to each other

- Swift: baseline with Swift CC
- 2-ToR Poseidon: 2 ToR switches support INT
- 4-ToR Poseidon: 4 ToR switches support INT
- Poseidon: all switches support INT

**Performance improves as more switches support INT feature.**

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Performance Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Swift CC</td>
<td>Baseline</td>
</tr>
<tr>
<td>2-ToR Poseidon</td>
<td>2 ToR switches support INT</td>
</tr>
<tr>
<td>4-ToR Poseidon</td>
<td>4 ToR switches support INT</td>
</tr>
<tr>
<td>Poseidon</td>
<td>All switches support INT</td>
</tr>
</tbody>
</table>

![Graph showing performance gain across different scenarios.](image)
Conclusion

• Poseidon algorithm uses quantitative per-hop INT:
  • **Decouples fairness from AIMD**
    • Gives a cluster of functions that can achieve fairness
    • Picks adaptive MIMD algorithm for outstanding performance
  • **Achieves max-min fairness**
    • Multi-hop congestion & reverse-path congestion
  • **Supports incremental deployment**
    • Performance improves when only ToR switches provide INT
• Poseidon is now open-sourced in ns-3 (developed based on the paper)
  • [https://github.com/Clark5/Poseidon](https://github.com/Clark5/Poseidon)