Host-pluggable RDMA Congestion Control

Vivek Kashyap

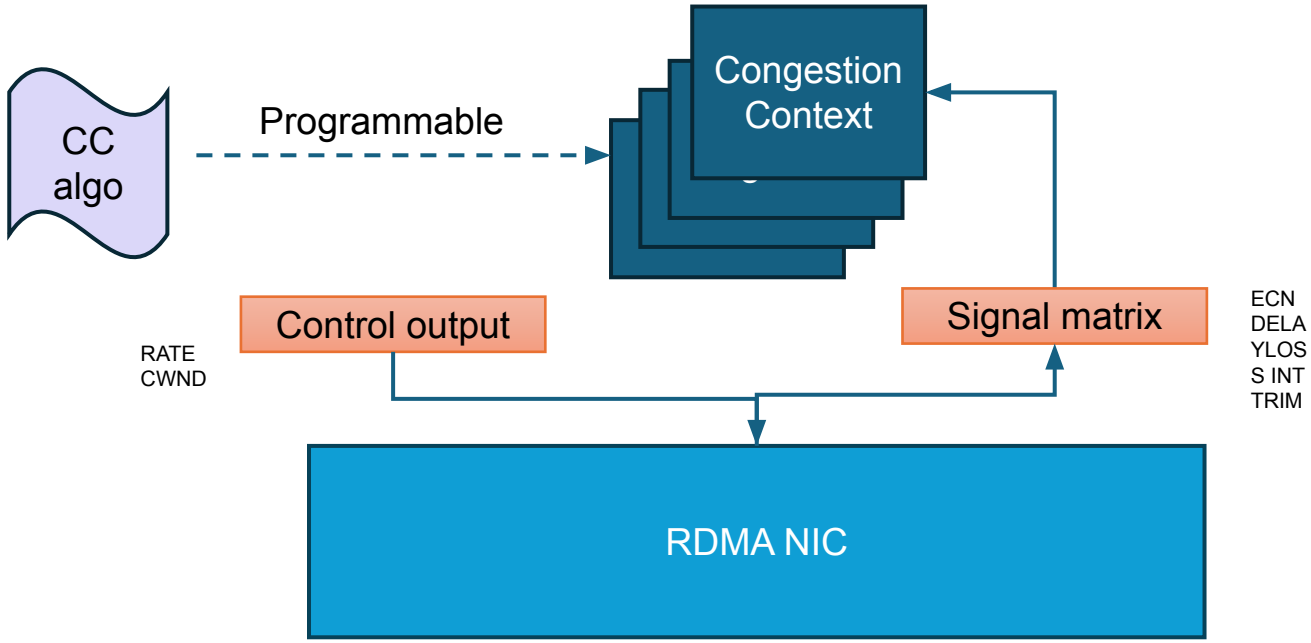
vivek.kashyap@intel.com

Programmable Congestion-control

- RDMA congestion control is fixed-function in the NIC
- Move congestion control into a host-accessible, pluggable control framework
  - Enable algorithms to be prototyped, tuned, compared, and evolved without requiring firmware or hardware changes.
  - Support policy-specific congestion behavior for AI/ML, storage, HPC, KV-cache movement, and front-end traffic.
- Identify the minimal endpoint interfaces needed for signal gathering, algorithm execution, and rate/CWND enforcement.
- Consider a generic framework on Linux for programmable RDMA CC

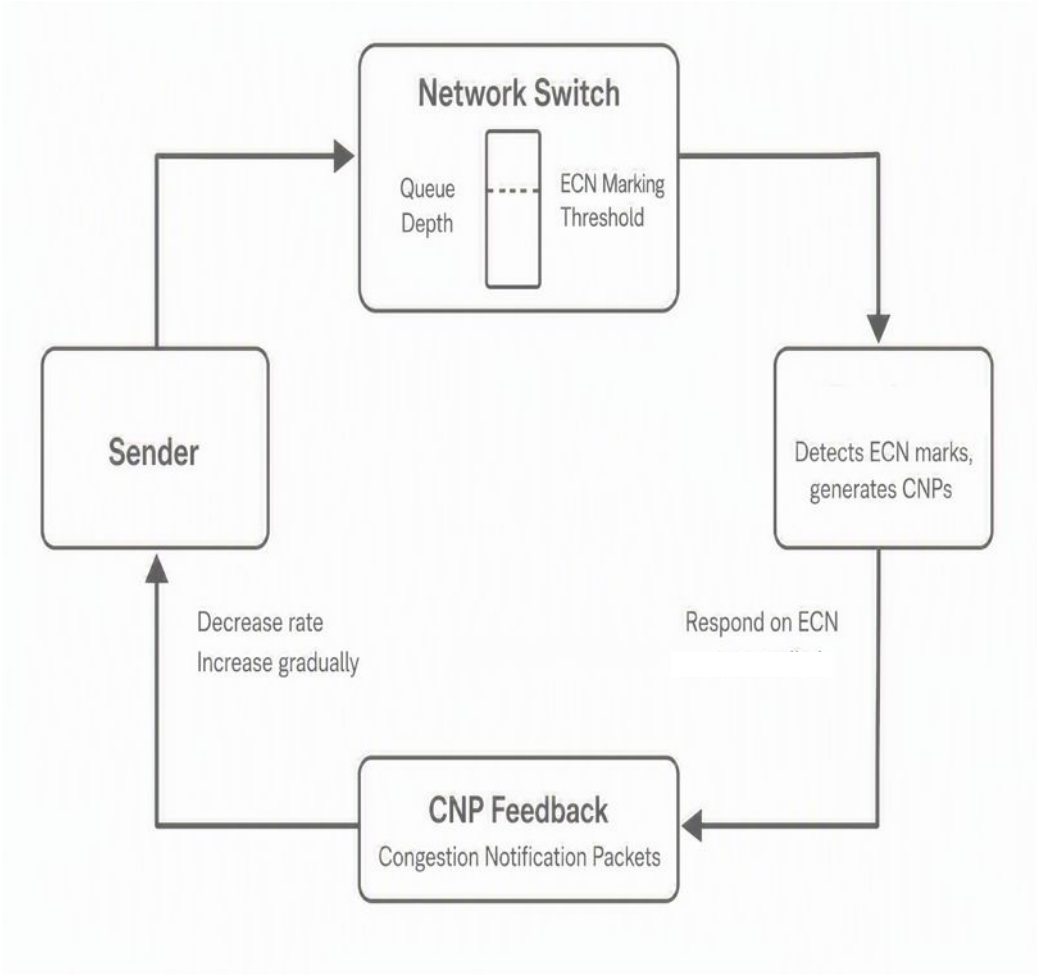
Congestion control loop

- A general congestion loop
  - Input signals -> congestion algorithm -> output constraints
  - Per flow or per congestion context

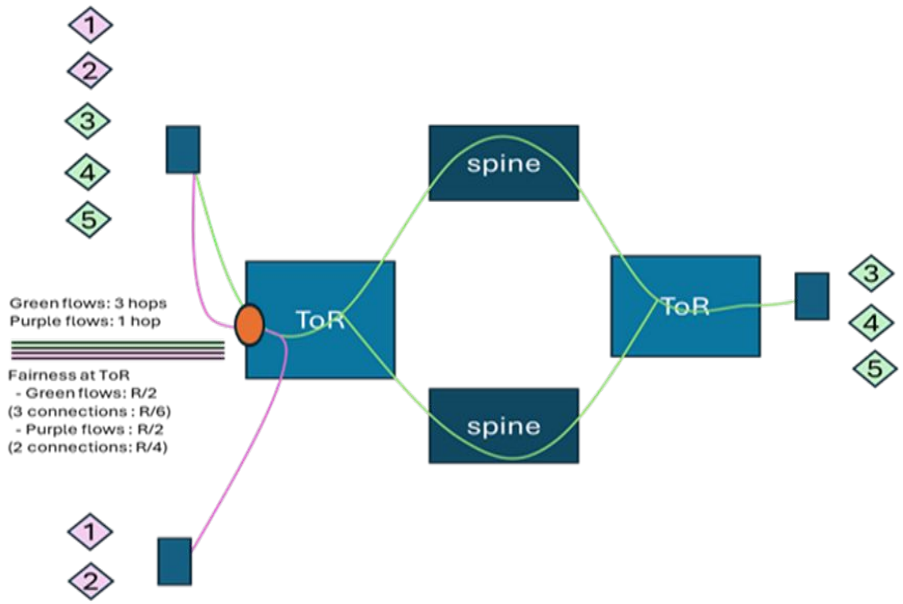


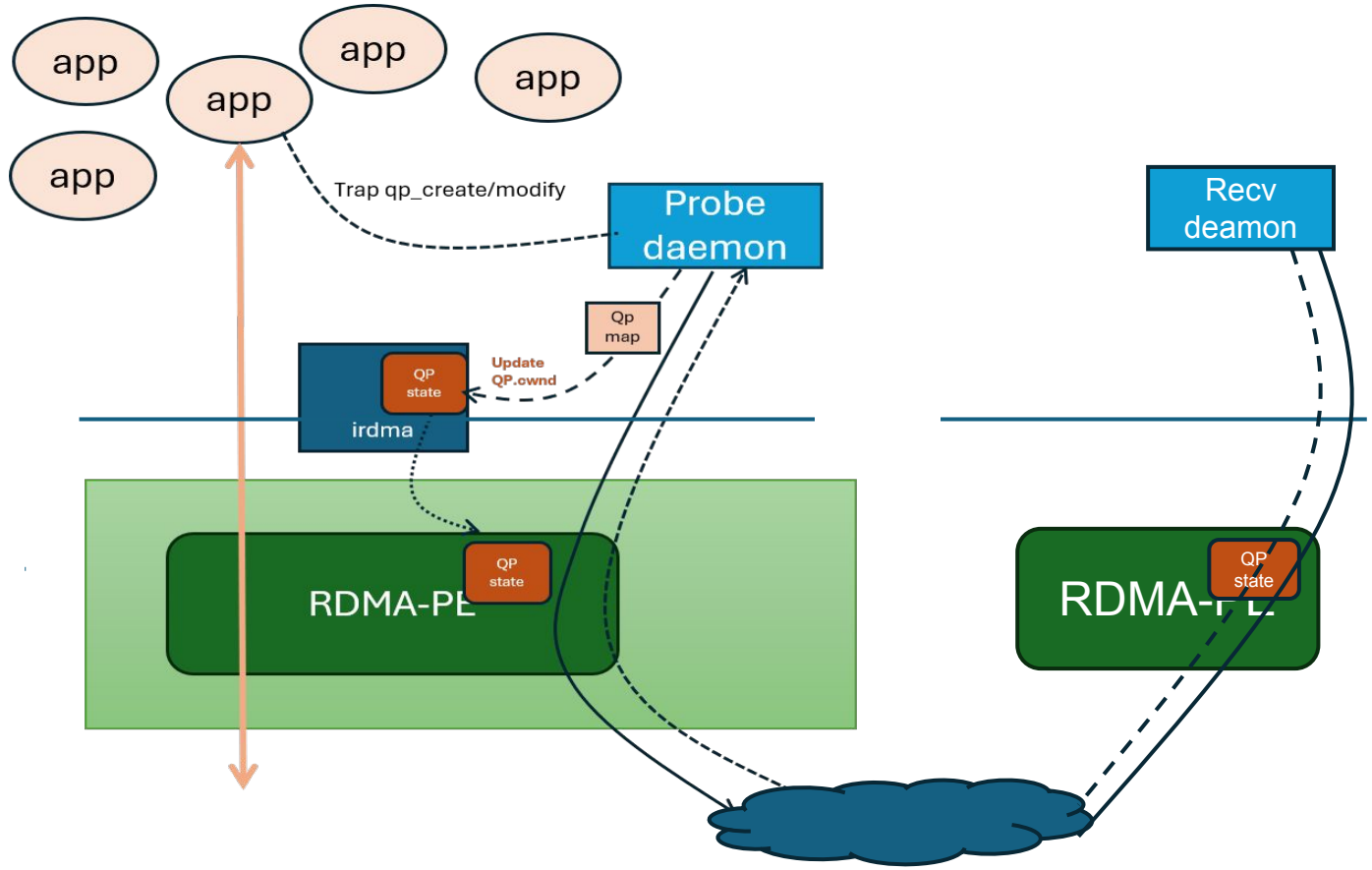
Example: DCQCN

- DCQCN is a rate-based congestion-control algorithm embedded in the RNIC
- Switch **ECN-marks,** the destination RNIC detects and sends **Congestion Notification Packets (CNPs)** back to the source
- The source RNIC uses CNP feedback to estimate congestion and set the new rate.
- DCQCN is commonly deployed with PFC in RoCEv2 fabrics



pRTT





- Common signals for a context
- Probes sent on a special QP or on LAN
  - Hardware timestamps
    - depends on hardware capabilities
- No PFC, no ECN, no CNPs

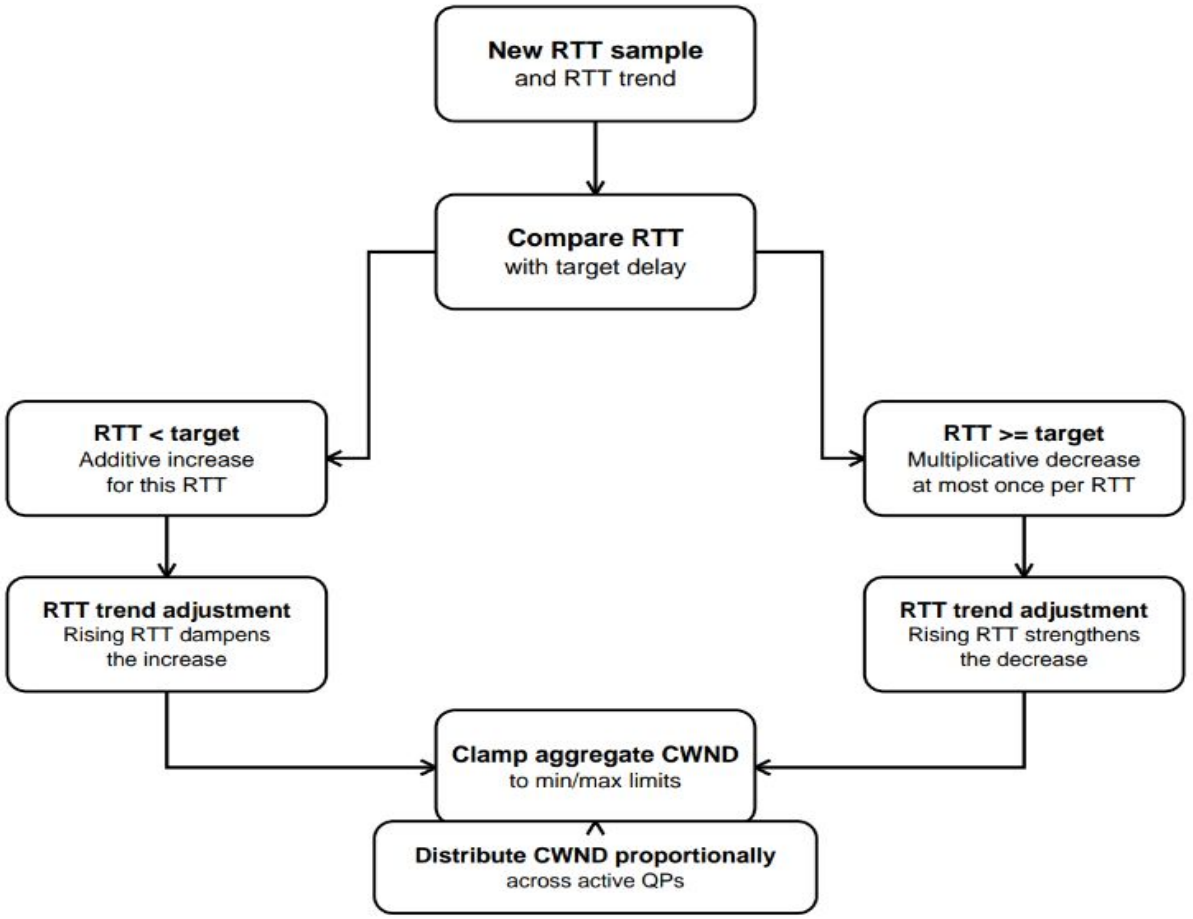
- Determine QP membership
  - Get signals per Congestion context (QP set)
- Calculate CWND
- For each QP member
  - Proportional rate update through the driver

probe RTT (pRTT): userspace RDMA CC

CC control loop	pRTT (user space congestion mgmt.)	DCQCN (Fixed function congestion mgmt.)
Congestion Context	QPs to same destination	QP
Signal Matrix	RTT	ECN
Input source	probe response h/w timestamp	CNPs
Congestion algorithm	Delay compared to target delay	DCQCN – responds to CNPs
CC output	Proportionally applied to QPs	Rate control on QP
Control method	Driver writes to QP state	Internal state updated within RNIC
Network	lossy	lossless

Algorithm: Calculate CWND

- Input: One probe-measured RTT and RTT trend per control interval (Gradier
- Below target: Additive increase
- At/above target: Multiplicatively decrease, limited to once per RTT
- Trend adjustment: Rising RTT damper increase or strengthens decrease
- Output: Clamp aggregate CWND and distribute it proportionally across active QPs



pRTT 3*4 : 1 Incast

Per-Flow Bandwidth Results

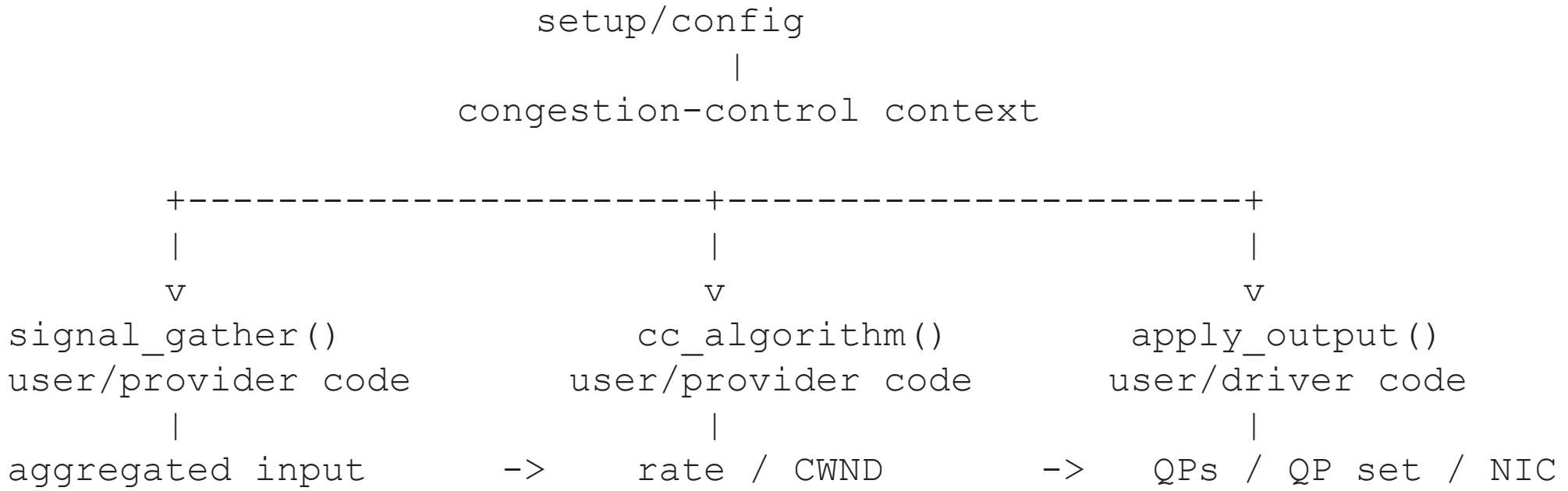
Flow #	Client	Port	Bandwidth (Gb/sec)
1	Client A	80001	8.01
2	Client A	80002	6.39
3	Client A	80003	9.10
4	Client A	80004	7.15
5	Client C	80005	6.95
6	Client C	80006	9.25
7	Client C	80007	7.55
8	Client C	80008	8.25
9	Client B	80009	6.01
10	Client B	80010	8.48
11	Client B	80011	9.51
12	Client B	80012	7.99

Individual flows vary due to start and QP write latencies

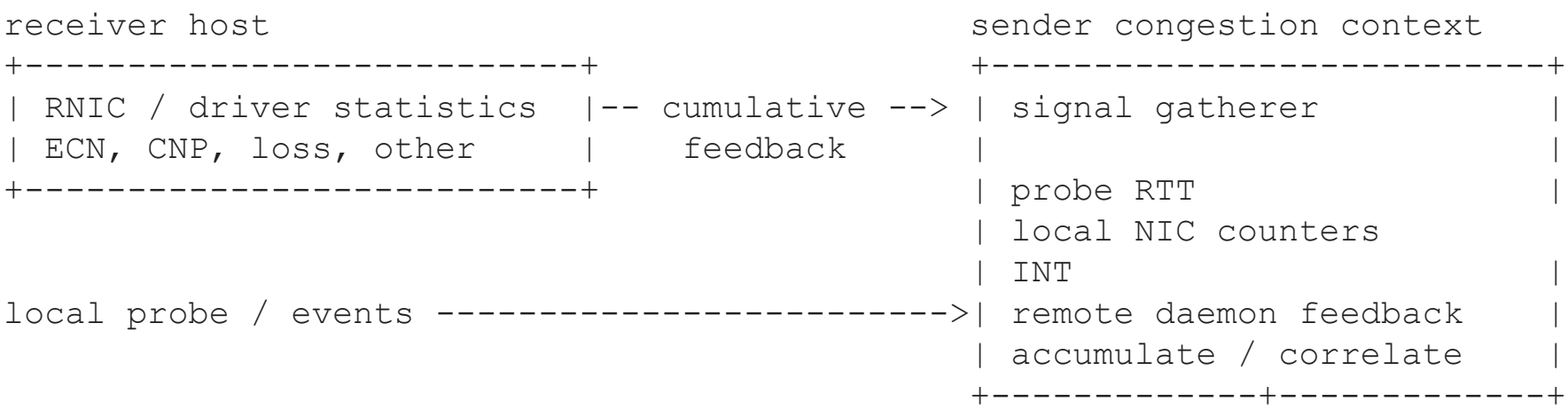
Per client aggregate is fair (approximately) – 30Gbps

From Prototype to Framework

- Prototype disables NIC-embedded CC. Uses special driver interfaces.
- Linux framework should retain the control-loop structure while replacing prototype specific coupling.
  - Basic control instance binds setup, congestion context, signal gatherer, algorithm, output applier.
  - The framework owns lifecycle, invocation, serialization, capability checks, and teardown.
  - Each implementation owns its signal semantics and device-specific enforcement.



signal_gatherer(): local and remote inputs



- The gatherer owns acquisition, accumulation, smoothing, freshness, and publication rules.
- Cumulative remote counters tolerate missed feedback
- The gatherer computes interval deltas.

Gatherer-to-algorithm execution

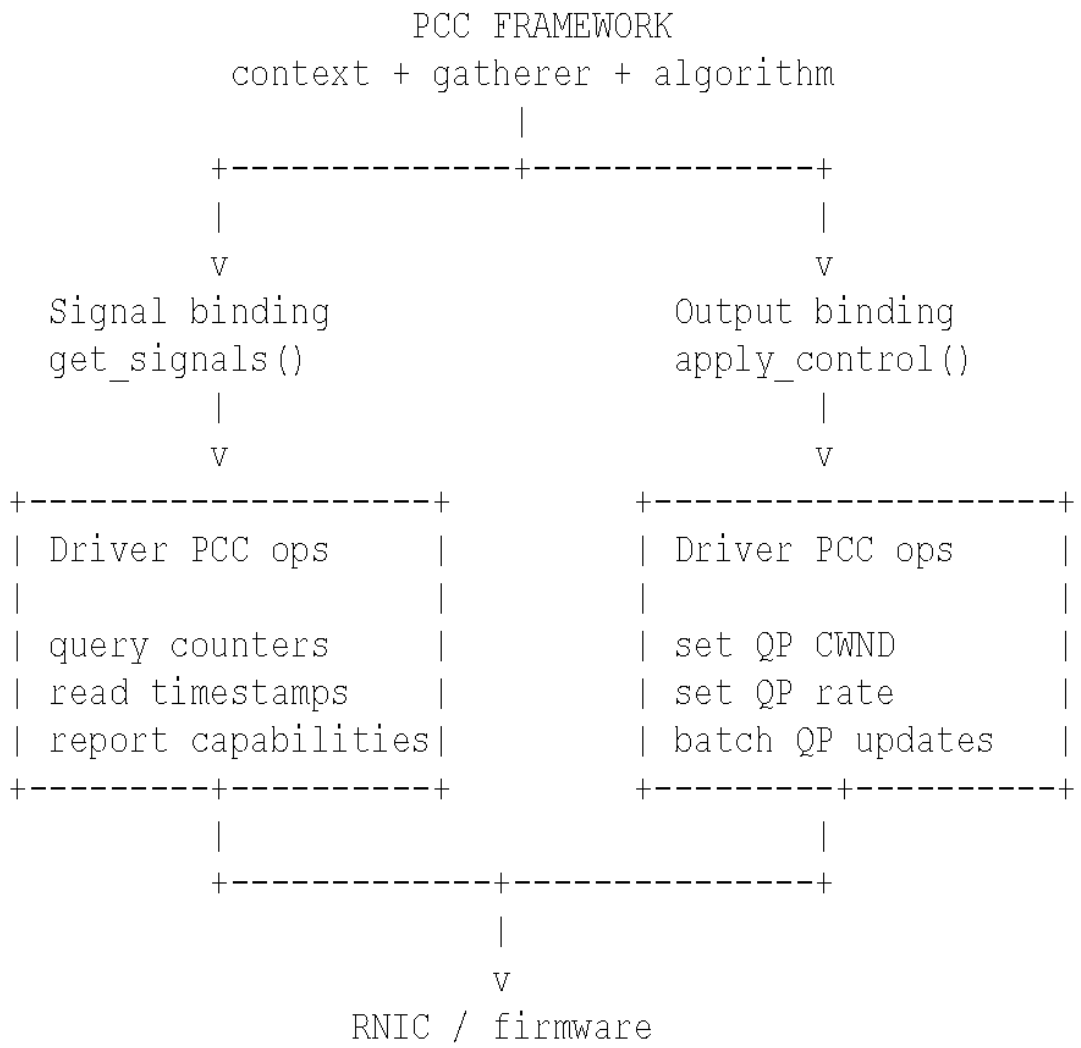
```
Option 1: inline                          Option 2: worker thread
------------------------------            ------------------------------
gatherer has input                        gatherer publishes latest input
call algorithm directly                   signal algorithm pthread/workqueue
low handoff latency                       gatherer remains unblocked
algorithm must not block                  stale inputs are coalesced

cc_algorithm(context, input, state) -> { no change | CWND | rate }
                                                    |
                                        Output-> write to QP state
```

- The framework supports one serialized algorithm invocation per context
- The gatherer receives raw events, and the algorithm receives one gatherer-prepared input.
- The algorithm is never invoked while the gatherer holds its update lock.

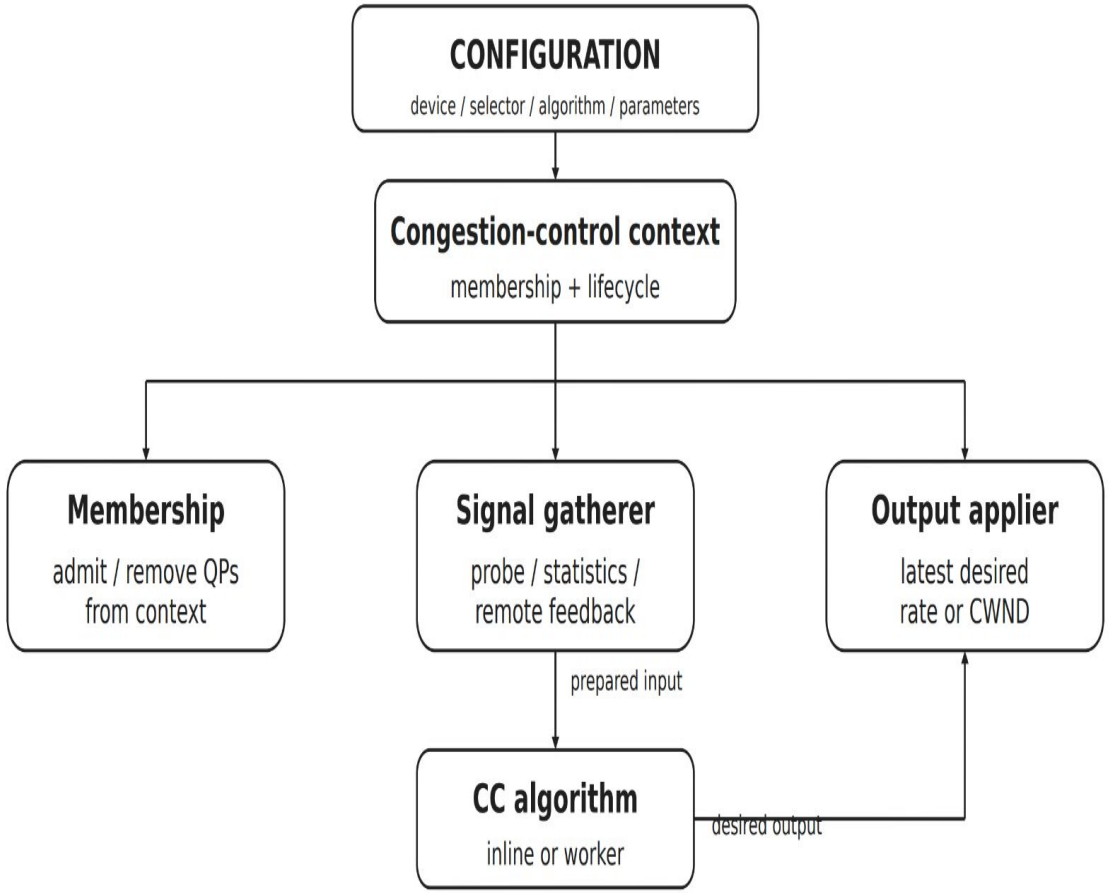
Framework mediated inputs

- Prototype uses modified driver and codes to it
- Possible extensions
  - driver_ops->get_signals(device, scope, requested_signals, &snapshot);
    - Signals matrix to be defined
  - driver_ops->apply_control(device, qp_set, PCC_CONTROL_CWND/RATE, value);



Exampe: pRTT as framework specification

Framework element	pRTT composition
setup/config	disable embedded CC; configure peers/NIC
context membership	QPs associated with destination
signal_gather	timestamped active probe
prepared input	RTT sample and metadata
algorithm state	srtt, gradient, target delay, aggregate CWND
algorithm	delay/trend controller
output	aggregate context CWND
apply_output	distribute and program per-QP CWND



Five ways to package framework

- Built into one daemon
  - Simplest first implementation; direct function calls.
- Userspace shared-object plugins (.so)
  - Load named components at startup with dlopen()/dlsym().
- Separate daemons or services
  - Process isolation: IPC, lifecycle, and latency cost.
- Kernel built-in code or loadable modules (.ko)
  - Close to QP lifecycle and driver events
- eBPF algorithm programs
  - Dynamic verified policy, still needs context, inputs, helpers, and outputs

Example Framework Packaging with pRTT

```
int main(int argc, char **argv)
{
        framework_init();   load_builtin_components();   load_plugins();
        load_configuration();
        create_configured_contexts();
        framework_run();
        framework_shutdown();
}
```

Context selection

membership = "destination-qps"
signal = "probe-rtt"
algorithm = "prtt"
output = "irdma-qp-cwnd"

Plugin responsibilities

probe-rtt: probe path, calculate RTT
prtt: RTT/trend, return aggregate CWND
irdma-qp-cwnd: obtain current members;
distribute/program CWND

Illustrative plugin descriptor

```
struct pcc_plugin_descriptor {
uint32_t abi_version;
const char *plugin_name;

const struct pcc_membership_ops *membership_ops;
const struct pcc_signal_ops *signal_ops;
const struct pcc_algorithm_ops *algorithm_ops;
const struct pcc_output_ops *output_ops;
};
```

Example: .so based plugin

PCC daemon startup
|
+-- dlopen("libpcc_probe_rtt.so")
| -> signal ops: "probe-rtt"
|
+-- dlopen("libpcc_prtt_algo.so")
| -> algorithm ops: "prtt"
|
+-- dlopen("libpcc_irdma_cwnd.so")
-> output ops: "irdma-qp-cwnd"

Configuration selects registered names - not function addresses.

Questions

Vivek.Kashyap@intel.com

backup

Practical Congestion Control Signals

Network Signals

- Loss: TCP Cubic, Reno
  - AIMD response
- Notification: DCTCP, DCQCN
  - ECN – receiver sends feedback to sender. Rate control
- Delay: Swift, TCP Vegas, TIMELY
  - Lagging indicator of congestion
  - RTT and Rate control
  - Swift: pacing for incast

Switch signals

- ECN: AQM – at egress
  - Leading indicator of congestion
- INT: HPCC
  - Get network information
    - Queue length, transmitted bytes
    - Link capacity, timestamp