1. Function APIs for Congestion Control Implementation

Our UDT-based PCC codebase adopts the same function APIs from Google QUIC, where a congestion control protocol is implemented as a class object:

- Class `PccSender` under src/pcc/pcc_sender.h implements PCC Allegro;
- Derived class `PccVivceSender` under src/pcc/pcc_vivace_sender.h implements PCC Vivace.

Each of those classes needs to implement several important functions. The two most important ones are:

- `OnPacketSent(sent_time, bytes_in_flight, packet_number, bytes, …)`

  This function is called each time a packet is sent. In PCC. This is where the sender stores the metadata corresponding to each sent packets:
  - `sent_time`: the timestamp when this packet is sent (can be used to calculate RTT gradient)
  - `bytes_in_flight`: number of bytes currently in flight (TCP uses this to throttle transmission)
  - `packet_number`: a strictly increasing identifier that distinguishes each packet
  - `bytes`: the size of the sent packet

-
  ```
  OnCongestionEvent(rtt_updated, rtt, bytes_in_flight, event_time,
                    acked_packets, lost_packets)
  ```

  This function is called when an ACK is received, or a timeout that signals packet lost is triggered. It is where the stored packet metadata is updated (e.g., RTT), and the rate control logic is initiated (when necessary).
  - `rtt_updated`: a boolean flag telling whether a new RTT sample is obtained
  - `rtt`: the latest RTT sample (if `rtt_updated` is true)
  - `bytes_in_flight`: the number of bytes currently in flight
  - `event_time`: the timestamp when this function is called
  - `acked_packets`: the vector of acked packets (including packet number and packet size)
  - `lost_packets`: the vector of lost packets (including packet number of and packet size)

2. PCC Code Structure

Fundamentally, PCC is based on a utility framework, which builds the causal relation between a sending rate and its corresponding achieved numeric utility. The sender sends at each sending rate for a monitor interval (lasts for at least one RTT). Then, when all packets belonging to a monitor interval is either acknowledged or lost, the sender calculates the interval's utility, based on which the sender may change the sending rate for the next monitor interval.

We illustrate the structure of our implementation in Figure 1. As shown in the figure, the two functions `OnPacketSent(.)` and `OnCongestionEvent(.)` provide the interface between the UDT transport library and our PCC codes. The PCC sender class has two member variables: `interval_queue_` and `utility_manager_`. The former is an object of class `PccMonitorIntervalQueue`. It maintains a deque queue of monitor intervals, each of which corresponds to a sending rate and stores metadata (such as the total bytes that are sent/acknowledged/lost and the packet RTT samples). When all packets belonging to one (or several, depending on the rate control algorithm) monitor interval(s) are either acknowledged or lost,

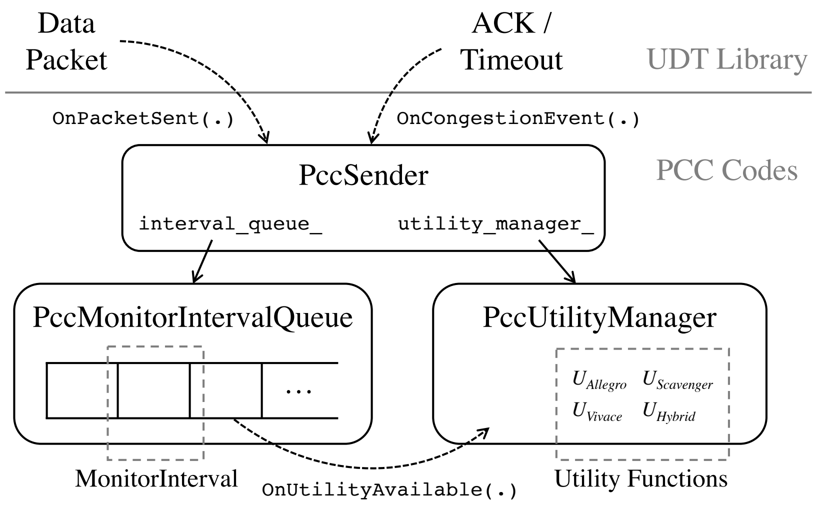
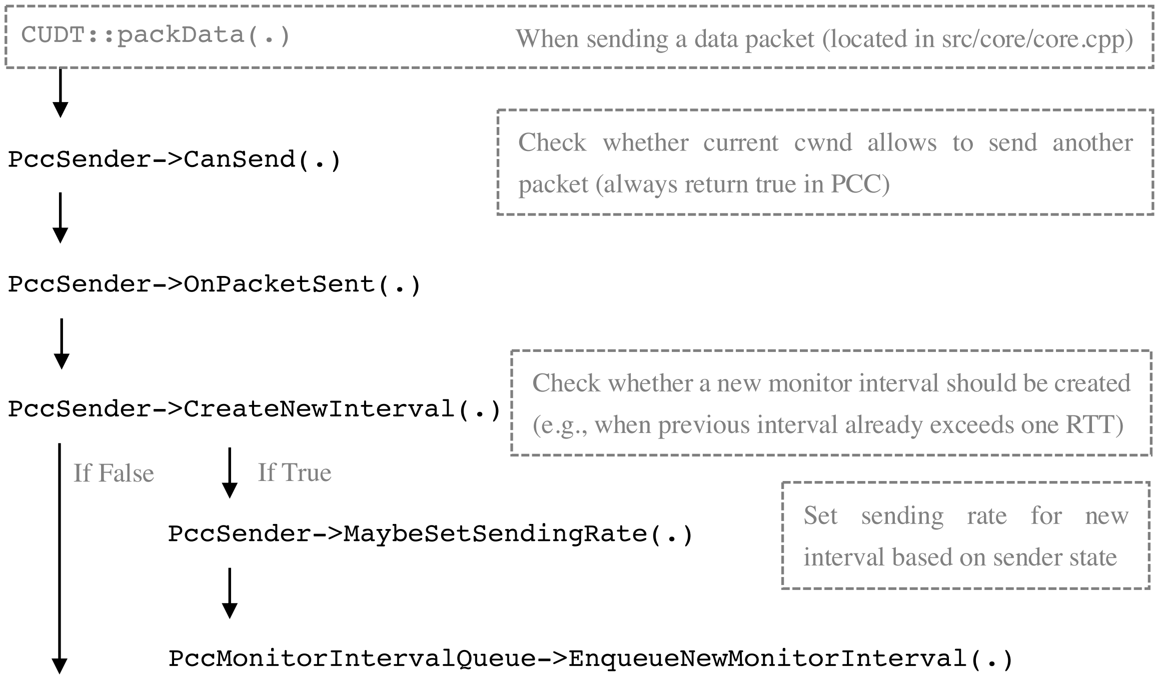


Figure 1. PCC Code Structure

the sender refers to `utility_manager_` to calculate utility value. Here, `utility_manager_` is an object of class `PccUtilityManager`, which includes a library of utility functions. Note that each utility function is constructed from carefully selected performance metrics (e.g., throughput, loss rate, RTT gradient). Before calculating interval utility, `utility_manager_` first computes those performance metrics from the interval's metadata.

3. Function Calling Hierarchy

At last, we detail the hierarchy of function calls specific to packet transmission and ACK reception, so as to help those interested utilize and improve our codes.



```
CUDT::packData(.)
PccSender->CanSend(.)
PccSender->OnPacketSent(.)
PccSender->CreateNewInterval(.)
    PccSender->MaybeSetSendingRate(.)
    PccMonitorIntervalQueue->EnqueueNewMonitorInterval(.)
```

When sending a data packet (located in src/core/core.cpp)

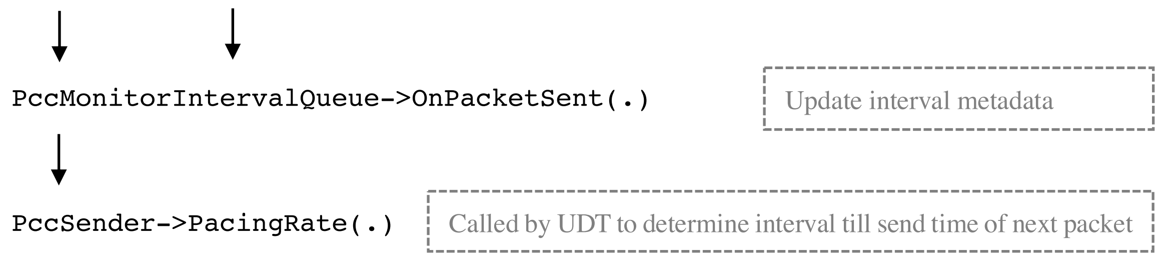
Check whether current cwnd allows to send another packet (always return true in PCC)

Check whether a new monitor interval should be created (e.g., when previous interval already exceeds one RTT)

If False

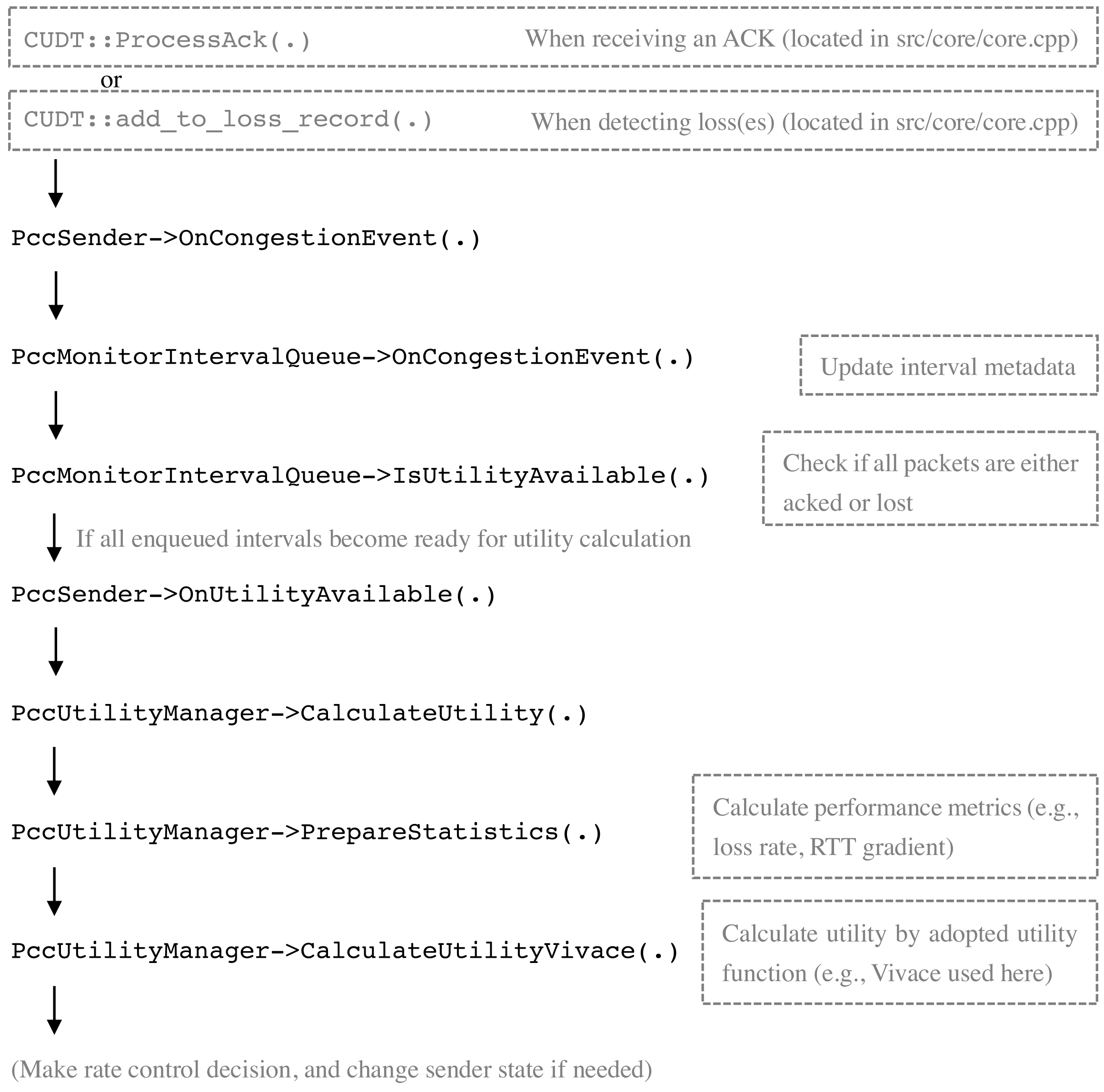
If True

Set sending rate for new interval based on sender state



`PccMonitorIntervalQueue->OnPacketSent(.)` — Update interval metadata

`PccSender->PacingRate(.)` — Called by UDT to determine interval till send time of next packet



`CUDT::ProcessAck(.)` — When receiving an ACK (located in src/core/core.cpp)

or

`CUDT::add_to_loss_record(.)` — When detecting loss(es) (located in src/core/core.cpp)

`PccSender->OnCongestionEvent(.)`

`PccMonitorIntervalQueue->OnCongestionEvent(.)` — Update interval metadata

`PccMonitorIntervalQueue->IsUtilityAvailable(.)` — Check if all packets are either acked or lost

If all enqueued intervals become ready for utility calculation

`PccSender->OnUtilityAvailable(.)`

`PccUtilityManager->CalculateUtility(.)`

`PccUtilityManager->PrepareStatistics(.)` — Calculate performance metrics (e.g., loss rate, RTT gradient)

`PccUtilityManager->CalculateUtilityVivace(.)` — Calculate utility by adopted utility function (e.g., Vivace used here)

(Make rate control decision, and change sender state if needed)

Specifically, in different sender states, the sender may require different number of monitor intervals before making a rate change decision. The function `OnUtilityAvailable(.)` is called only when all intervals in the monitor interval queue (for one rate change decision) become available (i.e., all packets are either acknowledged or lost). It is implemented as a delegate function, so that the monitor interval queue can directly call this function, while the actual execution (including the rate control logic and change of sending rate) happens within the sender class.