Note on Zenoh

Hsin-Jui, Chen

December 13, 2024

Contents

1 Tracing Zenoh

There are multiple ways to efficiently trace the Zenoh program using `uftrace`, one may follow these steps:

```
/* modify the toolchain */
$ cd $zenoh-repo/
$ vim rust-toolchain.toml                                // modify to "nightly"

/* install uftrace and compile zenoh */
$ sudo apt install uftrace
/* build zenoh using the following command lines */
$ export RUSTFLAGS=\  // Leading spaces are not allowed in the next line
"-Z instrument-mcount -C passes=ee-instrument<post-inline>"
$ cargo build --all-targets
$ uftrace record ./target/examples/z_pub
$ uftrace {replay,graph,tui}
```

To successfully compile the program and enable the program to traced in instruction-level. One may need to find out the right environment variables given to the compiler. I solve my problem with the comment in this reference, which provides some useful guidance for identifying the correct compile flag.

Another way to trace Zenoh programs is using the macros `tracing::info!()` and `tracing::debug!()` provided in crate `tracing`. These macros provide the same functionality as the `println!()` macro, however, it can separate the output into multiple levels of information. The following command is an example of running Zenoh program with `tracing` crate utilities.

```
$ RUST_LOG=debug cargo run --example example_name
/* You may also build and run */
$ cargo build --example
$ RUST_LOG=debug ./target/release/examples/example_name
```

To whom interested in how it works, you may check out the comment of the function `zenoh::try_init_log_from_env()` and its implementation. [1]

2 Case study: The `z_pub` example

```
# main() in examples/examples/z_pub.rs
let session = zenoh::open(config).await.unwrap();
```

The function `zenoh::open(config)` [2] returns an `OpenBuilder<TryIntoConfig>` instance, a structure that implements the `IntoFuture` trait. Therefore, the `.await` operator asynchronously waits for the `wait()` [3] function to complete.

[1] Line 31 in the file commons/zenoh-util/src/log.rs.
[2] Line 2900 in zenoh/src/api/session.rs
[3] Line 2954 in zenoh/src/api/session.rs

Then `wait()` function attempts to extract the config and passes it to the function `Session::new(config).wait()`[4]. This method first collects the topics it suscribes to and publishes. It then initializes a new runtime by

```
# Session::new() in zenoh/src/api/session.rs
let mut runtime = RuntimeBuilder::new(config);
let mut runtime = runtime.build().await?;
```

Let's dive into the details of `runtime` and `RuntimeBuilder::build()`[5]. This function first sets up a router (responsible for managing the routing table), a transport manager (handling network protocols), and a notifier (notifying when subscribed topics are received). It then initializes the runtime state with the router, transport manager, and notifier. For reference, a brief snippet of the code is shown below:

```
# RuntimeBuilder::build() in zenoh/src/net/runtime/mod.rs
let router = Arc::new(Router::new(zid, whatami, hlc.clone(), &config)?);
let tm_builder = TransportManager::builder()
    .from_config(&config)
    .await?
    .whatami(whatami)
    .zid(zid);
let transport_manager = tm_builder.build(handler.clone())?;
let config = Notifier::new(config);
let runtime = Runtime {
    state: Arc::new(RuntimeState {
        ...                                             // ommited
    }),
};
```

After that, it makes the process listen for incoming messages by starting an asynchronous notifier that listens for `"connect/endpoints"` with the runtime and updates peers as needed. However, I do not fully understand this part of the code, so I am only sketching the main ideas here. Once the runtime is built, the session is then initialized with the runtime.

```
# Session::new() in zenoh/src/api/session.rs
let mut session = Self::init(
    runtime.clone(),
    aggregated_subscribers,
    aggregated_publishers,
)
.await;
```

It appears that each session must be associated with a runtime; however, multiple sessions can share the same runtime. Within `Session::new()`, the `Session::init()`

[4] Line 955 in zenoh/src/api/session.rs
[5] Line 126 in zenoh/src/net/runtime/mod.rs

function is invoked, which subsequently calls `router.new_primitives()`[6]. This function adds a new `FaceState` entry to the routing table (`tables.faces`) and generates a unique `fid`. For reference, the relevant code snippet is shown below:

```
# Router::new_primitive() in zenoh/src/net/routing/router.rs
let fid = tables.face_counter;
tables.face_counter += 1;
let newface = tables
    .faces
    .entry(fid)
    .or_insert_with(|| {
        FaceState::new(
            fid,
            zid,
            WhatAmI::Client,
            #[cfg(feature = "stats")]
            None,
            primitives.clone(),
            None,
            None,
            ctrl_lock.new_face(),
        )
    })
    .clone();
tracing::debug!("New {}", newface);
for (key, val) in tables.faces.iter() {
    tracing::debug!("key: {key} val: {val}");
}
```

From this, we can observe that the face ID (`fid`) is determined by a counter. Therefore, the observation mentioned in the meetings on 11/1 and 11/8, that `fid`s are inconsistent across Zenoh processes, is now clear and understandable.

After the `faces` is constructed, it is cloned using `Arc::clone()` and inserted into a `Face` structure, which is then returned. Subsequently, `admin::init(&session)` is called within `Self::init()`. However, the functionality of this method remains unclear to me. Once this step is complete, the session is successfully initialized and returned from `Self::init()` (`Session::init()`).

Next, the `Runtime::start()` function is invoked [7]. This function determines the appropriate method to call based on the value of runtime.WhatAmI and starts the runtime accordingly. In the `z_pub` example, the value of `WhatAmI` is set to `Peer`, resulting in the invocation of the `Runtime::start_peer()` method[8]. In the `Runtime` method `Runtime::start_peer()`, the following operations are performed:

[6] Implemented at line 70 in zenoh/src/net/routing/router.rs
[7] Line 119 in zenoh/src/net/runtime/orchestrator.rs
[8] Line 171 in zenoh/src/net/runtime/orchestrator.rs

1. Bind listeners (`self.bind_listeners(&listeners).await?;`): This step appears to create multiple listener threads to handle connections on different threads.

2. Connect to peers (`self.connect_peers(&peers, false).await?;`): Establishes connections to other peers.

3. Enable scouting (if configured): If scouting is enabled in the configuration, the runtime will start the scouting process.

When binding listeners, the critical functions in the call tree are as follows:

```
Runtime::bind_listeners(&listeners)
  Runtime::bind_listeners_impl(listeners)
    Runtime::add_listener(&self, listener: EndPoint)
      Runtime::manager().add_listener(endpoint)
        TransportManager::add_listener_unicast(endpoint)
            TransportManager::new_link_manager_unicast(...)
            LinkManagerUnicastTcp::new_listener(endpoint)
              LinkManagerUnicastTcp::new_link_inner(...)
```

Then the TCP stream socket is returned.

3 On `io/zenoh-transport`

3.1 Basic Ideas of Transmission Pipeline

After some seniors advised me to switch my focus to studying the zenoh trasport, I start reading the codes in `io/zenoh-transport`, and I realized that this part might be more useful for understanding how Zenoh handles message transport.

I began by examining the code in `io/zenoh-transport/src/common/pipeline.rs`, which implements the pipeline transmission mechanism in Zenoh. The following note is written with respect to the earlier 1.0.0 beta version, it might have some differences now. This code establishes a MPSC (**Multiple Producer, Single Consumer**) pipeline. Consequently, the pipeline is divided into two stages: `StageIn` and `StageOut`.

For a data producer, it must insert data into a `WBatch` (short for *write batch*). The `WBatch` structure is defined in `io/zenoh-transport/src/common/batch.rs` as follows:

```rust
pub struct WBatch {
    pub buffer: BBuf,           // The buffer to perform the batching on
    pub codec: Zenoh080Batch,                         // The batch codec
    /* It contains 1 byte to signal whether the batch is compressed */
    pub config: BatchConfig,
    #[cfg(feature = "stats")]         // Statistics related to this batch
    pub stats: WBatchStats,
}
```

Essentially, the `WBatch` strucuture serves as an in-memory buffer for storing serialized data. To implement the MPSC pipeline, multiple `WBatch` structures are organized into a `RingBuffer`. This design allows multiple producers to insert data into different batches concurrently. However, the implementation is more complex than it appears.

Next, we explain the key ideas behind the `push_network_message()` function in the `StageIn` implementation.

1. The function first attempts to obtain a `WBatch` by following these steps:

(**a**) Acquire the lock associated with the `StageIn`.

- The thread acquires a lock from `StageInMutex` to check if a `WBatch` is available.
- If an available `WBatch` is found in the `StageInMutex`, the buffer is successfully obtained. The `WBatch` is then moved out of the `Option<WBatch>`, signaling to other threads that this `WBatch` needs to be refilled. [9]

(**b**) Refill Check.

- If no `WBatch` is available, the function attempts to refill by checking for remaining `WBatch` structures in the ring buffer. If successful, the thread obtains a buffer from the ring buffer.

(**c**) Conditional Wait.

- If the ring buffer also lacks an available `WBatch`, the thread must wait for a consumer to process a batch and notify the thread. This involves a conditional wait mechanism. Additionally, certain scenarios may require further considerations. For instance, if some messages are allowed to be dropped after a deadline, the thread may wait only until the deadline expires.

The following piece of code implements this idea:

```
macro_rules! zgetbatch_rets {
  ($fragment:expr, $restore_sn:expr) => {
    loop {
      match c_guard.take() {
        Some(batch) => break batch,
        None => match self.s_ref.pull() {
          Some(mut batch) => {
            batch.clear();
            self.s_out.atomic_backoff.first_write.store(
                LOCAL_EPOCH.elapsed().as_micros() as MicroSeconds,
                Ordering::Relaxed);
            break batch;
          }
          None => {
            drop(c_guard);
            match deadline_before_drop {
```

[9] Initially, I thought that moving the `WBatch` out of the`Option<T>` was to prevent other threads from writing to the same `WBatch`. Thanks to those seniors for pointing out my misunderstanding.

```
              Some(deadline) if !$fragment => {
                if !self.s_ref.wait_deadline(deadline) {
                  $restore_sn;
                  return false
                }
              }
              _ => {
                if !self.s_ref.wait() {
                  $restore_sn;
                  return false;
                }
              }
            }
            c_guard = self.mutex.current();
          }
        },
      }
    }
  };
}
```

2. After obtaining the buffer, the function attempts to write the message into it. However, the acquired `WBatch` may not be empty, and the message might not fit into the remaining space. In such cases, the function releases the original `WBatch` and tries to obtain an empty `WBatch` using the same procedure described above.

3. If an empty `WBatch` still cannot accommodate the entire message, the function will iteratively obtain additional `WBatch` structures and fragment the message until it is fully processed.

Note that there is a distinction between `NetworkMessage` and `TransportMessage`: the former represents the data being transported, while the latter is used for protocol and flow control. As a result, a network message may be larger and may not fit into a single `WBatch`. However, in `push_transport_message()`, fragmentation handling is unnecessary. To implement the concepts described above, the following structures are designed to manage the `StageIn` and `StageOut` pipelines.

```
/* StageIn */
struct StageInRefill {
    n_ref_r: Waiter,
    s_ref_r: RingBufferReader<WBatch, RBLEN>,
}
struct StageInOut {
    n_out_w: Notifier,
    s_out_w: RingBufferWriter<WBatch, RBLEN>,
    atomic_backoff: Arc<AtomicBackoff>,
}
```

```rust
struct StageInMutex {
    current: Arc<Mutex<Option<WBatch>>>,
    priority: TransportPriorityTx,
}
/* StageOut */
struct StageOutIn {
    s_out_r: RingBufferReader<WBatch, RBLEN>,
    current: Arc<Mutex<Option<WBatch>>>,
    backoff: Backoff,
}
struct StageOutRefill {
    n_ref_w: Notifier,
    s_ref_w: RingBufferWriter<WBatch, RBLEN>,
}
```

We can observe that the `StageInRefill` structure holds the information, methods, and implementations necessary for waiting on a `WBatch` to be consumed. On the other hand, the `StateInOut` structure stores the information required to notify the consumer threads.

3.2 Backoff Mechanism in Transmission Pipeline

As I mentioned in the previous meeting, there are certain backoff mechanisms in the transmission pipeline. Initially, I believed these mechanisms were designed for congestion control. However, after further study, I realized this assumption was incorrect.

The actual idea is to avoid consuming a `WBatch` if it contains only a small amount of data. Backoff mechanisms are adopted to prevent the constant writing of small batches. Specifically, if only a few bytes are written to a `WBatch`, there is no immediate need to consume it. Waiting for additional bytes to be written can make the transmission more efficient by reducing communication overhead, such as headers.

Let me elaborate further. Recall that both the consumer and producer maintain their own queues for available `WBatch` instances. For the consumer, this queue (a ring buffer) holds the `WBatch` instances that are ready to be sent out. For the producer, its queue contains `WBatch` instances that are empty (but may not necessarily be clean). The use of these two queues offers an advantage: it limits the maximum number of pending WBatch instances and serves as a form of resource control.

For the consumer, if there is a `WBatch` in its queue, it simply pulls the batch and sends it out over the link. If a `WBatch` is present in the consumer's queue, it indicates that the batch is either full or needs to be sent immediately. In such cases, there is no need to wait. However, if the consumer queue is empty, the consumer might consider consuming the currently active `WBatch` in `StageIn`. To avoid frequently sending out `WBatch` instances with only a small amount of data, the consumer waits until one of the following conditions is met:

1. A `WBatch` is added to the consumer queue (this will be notified by producers).
2. The `WBatch` in `StageInMutex` contains data exceeding a predefined threshold (this will be notified by producers as well).
3. The backoff timeout is reached.

In the first case, a `WBatch` becomes ready while the consumer is waiting. In the second case, the WBatch is nearly full and can be sent out efficiently. In the third case, no new data is written to the current `WBatch`, indicating that waiting further is unnecessary. Next, we will examine the code to provide additional explanations.

```
struct AtomicBackoff {
    active: CachePadded<AtomicBool>,
    bytes: CachePadded<AtomicBatchSize>,
    first_write: CachePadded<AtomicMicroSeconds>,
}
```

This structure is shared by both the producer and consumer. The three fields in the structure have the following meanings:

1. `active`: Indicates whether the consumer is actively in backoff state.
2. `bytes`: Stores the number of bytes written to the current `WBatch`.
3. `first_write`: Stores the timestamp of the first message written to the current `WBatch`. This field is used to ensure that no message waits for too long.

The following code demonstrates how these ideas are implemented on the producer side.

```
impl StageInOut {
    #[inline]
    fn notify(&self, bytes: BatchSize) {
        self.atomic_backoff.bytes.store(bytes, Ordering::Relaxed);
        if !self.atomic_backoff.active.load(Ordering::Relaxed) {
            let _ = self.n_out_w.notify();
        }
    }
    #[inline]
    fn move_batch(&mut self, batch: WBatch) {
        let _ = self.s_out_w.push(batch);
        self.atomic_backoff.bytes.store(0, Ordering::Relaxed);
        let _ = self.n_out_w.notify();
    }
}
```

In the function `notify()`, the number of bytes written is first stored. If the consumer is in a backoff state (namely, the active field is true), the function notifies the consumer, waking it up.

For the method `move_batch()`, it pushes the current `WBatch` to the consumer queue. Since the current `WBatch` is being moved out, the number of bytes must be reset to 0. After this, the function notifies the consumer.

On the consumer side, the structure `StageOutIn` has two methods, `try_pull()` and `try_pull_deep()`, which implement the functionality for pulling messages either from the queue or from the current `WBatch`. The following pseudocode demonstrates the workflow for pulling a `WBatch` on the consumer side.

Algorithm 1 The implementations of `try_pull()` and `try_pull_deep()` functions

```
1:  function TryPull( )
2:      if the consumer queue is not empty then
3:          set the backoff state to inactive            // it might be backoffing previously.
4:          return The WBatch in the queue
5:      else
6:          TryPullDeep( )                               // Try to pull current WBatch.
7:
8:  function TryPullDeep( )
9:      pull ← (backoff is active)    // Check whether it was already in backoff since last pull
10:     backoffTime ← 0
11:     if not pull then              // The consumer is already backoffed in previous pulls
12:         pull←(new_bytes == old_Bytes)   // Check whether some new bytes are added
13:     if not pull then                    // There are some bytes added by the producer
14:         diff ← elapsed time since the first write to the WBatch
15:         if diff ≥ threshold then
16:             pull ← True
17:         else
18:             backoffTime ← threshold - diff
19:     if pull then                                     // Ready to pull the current WBatch.
20:         acquire lock
21:         set backoff state to inactive
22:         if consumer queue is not empty then          // In case the queue is not empty
23:             return WBatch from the queue
24:         else if current WBatch does not exist then   // StageIn does not refill yet
25:             return None
26:         else                                         // Pull the current WBatch
27:             return current WBatch
28:     else               // The requirements of pulling current WBatch does not meet
29:         set backoff state to active
30:         return backoffTime                           // Make consumer wait for some time
```

This realization has made me reconsider some of my previous ideas about the transmission pipeline. It appears that certain assumptions I made were incorrect, which could have subtle (though likely negligible) impacts on the implementation of the `qstats` feature for per-pipeline profiling.

For example, one statistic of interest might be the average queuing delay of messages or `WBatch` instances. Suppose we aim to profile the queuing delay of `WBatch` instances. Intuitively, we might record the time when a `WBatch` is placed into the queue and then calculate the elapsed time when the same `WBatch` is pulled from the queue.

Consider a scenario where a message is written to a new `WBatch`, but the queue is never consumed. In this case, the `WBatch` holds data, yet no queuing delay is recorded, which feels conceptually inconsistent.

4 Profiling Zenoh

4.1 Official Implementation of the `stats` Feature

In this section, I study the file `io/zenoh-transport/src/common/stats.rs`, and I am going to briefly expalin how Zenoh team implement their statistics analysis tools.

Before we get started, I will first show how we can utilize these existing implementations. First, to enable this, you may need to enable the `stats` features when compiling, you may do it in many ways, for instance, the following compiling commands all work.

```
$ cargo run --example <example_name> --features stats
/* it is also possible separately build and run */
$ cargo build --example <example_name> --features stats
$ ./target/debug/<example_name>
```

To get famaliar with the `TransportStats` structure, I first implement printing out the statistics on dropping, that is, as soon as the `TransportUnicast` drop, then it will print out the statistics. Precisely speaking, I implement the `Drop` trait for the structure `TransportUnicastUniversal` as the following code listing. [10]

```
impl Drop for TransportUnicastUniversal {
    fn drop(&mut self) {
        tracing::info!(
            "{}",
            self.stats.clone().report().openmetrics_text()
        );
    }
}
```

Since the implementation is written in a complicated macro, I decide to expand the macro to have better understand on this complicated macro. I use the following commands to expand the macro.

```
$ cargo install cargo-expand
$ cd io/zenoh-transport/src
$ cargo expand common::stats --features stats > common/stats_expand.rs
```

We can see that it actually implement some increase counter methods and getter methods, this idea gives me some insights to build the equivalent of `qstats`.

[10] I implement this trait in the file io/zenoh-transport/src/unicast/universal/transport.rs.

4.2 Documentation for the `qstats` Feature

This subsection I am trying to build the `qstats` feature by imitating the `stats` feature after macro expansions. The code is available at this repository. I give definitions and implement some methods of the structures `QueueStats` and `QueuesStatsReport` in the file `io/zenoh-transport/src/common/qstats.rs`.

```rust
pub struct QueueStatsReport {
    priority: usize,
    pub avg_qsize: f64,
    pub droprate: f64,
    pub avg_qdelay: f64,
    pub mcnt: usize,
}

pub struct QueueStats {
    queue_counter: AtomicUsize,    // current number of queueing messages
    priority: usize,                         // the priority of this queue
    qsize: Arc<Mutex<Vec<usize>>>,        // recorded numbers of messages
    dropped: AtomicUsize,                 // the number of dropped messages
    tried: AtomicUsize,                     // the number of tried messages
    queueing_delay: Arc<Mutex<Vec<usize>>>, //  recorded queueing delays
}
```

These are the definitions of the structures, and we can see that `QueueStatsReport` will record the `QueueStats` at some point, while `QueueStats` structure continuously record the statistics. The main statistics we are concerned about are average queue size (`qsize`), number of dropped messages (`dropped`), drop rate due to queueing (`dropped/ tried`), and average queueing delay (average of `queueing_delay`).

The structure `QueueStats` have the following methods:

```rust
impl QueueStats {
    pub fn new(priority: usize) -> Self {}
    pub fn record_qsize(&self) {}
    pub fn push_qdelay(&self, delay: usize) {}
    pub fn inc_qcnt(&self) {}
    pub fn dec_qcnt(&self) {}
    pub fn inc_dropped(&self) {}
    pub fn inc_tried(&self) {}
    pub fn report(&self) -> QueueStatsReport {}
}
```

These methods have the following purposes:

1. `new()`: create a new `QueueStats` for priority level `prioirty`.
2. `record_qsize()`: when producer trying to push message, it will first need to record the number of messages in the queue. Hence the method need to first load the counter then push to `self.qsize`.

3. `push_qdelay()`: when a consumer pull a message, it will calculate the elapsed time since the message written to a `WBatch`.
4. `inc_qcnt()`, `dec_qcnt()` when a message is pushed or pulled, we need to modify the counter.
5. `inc_dropped()`, `inc_tried()` when a message is pushed, we need to change the counter of dropped and tried messages.
6. `report()` generate a `QueueStatsReport` structure that contains the current statistics. For the average queue size and average queueing delay, I choose to implement the exponential moving average formula, with $\alpha = 0.1$.

Modifications on `batch.rs`. There are modifications needed to be made to `batch.rs` and `pipeline.rs`. For `batch.rs`, we add the following to `encode()` [11] methods. We use the conditional compilation flag `#[cfg(feature = "qstats")]` to ensure the modifications are effective only when compiling with `qstats` feature.

```rust
struct WBatch {
    /* some fields are omitted */
    #[cfg(feature = "qstats")]
    pub time: Vec<Instant>,
}
impl Encode<...> for &mut WBatch {
    fn encode(...) -> Self::Output {
        /* some lines are omitted */
        let res = self.codec.write(&mut writer, x);
        #[cfg(feature = "qstats")]
        if res.is_ok() {
            self.time.push(Instant::now());
        }
        res
    }
}
```

When a message or a fragment buffer is successfully written to a `WBatch`, we add an instant (time) to `WBatch.time`. Then, when this `WBatch` is pulled, it will calculate queueing delay of these messages and fragments.

Modifications on `pipeline.rs`. In this paragraph, we only list important modifications. The consumer and the producer both will have a list of `QueueStats`.

```rust
struct StageIn {
    /* some fields are omitted */
    #[cfg(feature = "qstats")]
    qstats: Arc<QueueStats>,
}
pub(crate) struct TransmissionPipelineProducer {
```

[11] Note that there are multiple implementations of `encode()`.

```
    /* some fields are omitted */
    #[cfg(feature = "qstats")]
    qstats_list: Arc<[Arc<QueueStats>]>,
}
pub(crate) struct TransmissionPipelineConsumer {
    /* some fields are omitted */
    #[cfg(feature = "qstats")]
    qstats_list: Arc<[Arc<QueueStats>]>,
}
```

Although the QueueStats structure for each queue will be access by consumer and producer simultaneously, it is not needed to use Mutex because the inner data structure of QueueStats is protected by atomic operations and mutexes. That is, the interfaces provide by QueueStats are already thread-safe. With the modifications above, we can now look at the actual changes in the consumer and the producer.

```
impl TransmissionPipelineConsumer {
  pub(crate) async fn pull(&mut self) -> Option<(WBatch, usize)> {
    while self.active.load(Ordering::Relaxed) {
      let mut backoff = MicroSeconds::MAX;       // Calculate the backoff
      for (prio, queue) in self.stage_out.iter_mut().enumerate() {
        match queue.try_pull() {
          Pull::Some(batch) => {
            #[cfg(feature = "qstats")]
            {
              let qstat = &self.qstats_list[prio];
              batch.time.iter().for_each(|t| {
                qstat.dec_qcnt();
                qstat.push_qdelay(t.elapsed().as_micros() as usize);
              });
            }
            return Some((batch, prio));
          }
          Pull::Backoff(deadline) => {
            backoff = deadline;
            break;
          }
          Pull::None => {}
        }
      }
      /* the remaining part is omitted */
    }
  }
}
```

We see that in the conditional compile block, for each Instant on the WBatch, we will calculate the delay and decrease the counter by the number of messages or fragments. The next part the changes on producer side.

```
impl TransmissionPipelineProducer {
    pub(crate) fn push_{network,transport}_message(
        &self,
        mut msg: NetworkMessage
    ) -> bool {
        /* some lines are ommited */
        let result = queue.push_network_message(
            &mut msg, priority,
            &mut deadline
        );
        #[cfg(feature = "qstats")]
        {
            let qstats = &self.qstats_list[idx];
            qstats.record_qsize();
            qstats.inc_tried();
            if !result {
                qstats.inc_dropped();
            }
        }
        result
    }
}
```

In this part, we need to record the queue size when a message is pushed and increase the counter of tried messages. If the result is not successful, the counter of dropped messages should also increase. The counter of queueing messages is added in the `zretok!()` macro defined in `StageIn` methods `push_network_message()` and `push_transport_message()` and the fragment part of `push_network_message()`.

4.3 Documentation for the Link Scheduler

In this subsection, I am going to explained how I abstracted the original scheduler into `Scheduler` trait.

```
trait SchedulerTrait {
    fn reset(&mut self);
    fn schedule(&mut self) -> ScheduleResult;
}
enum ScheduleResult {
    Some((WBatch, usize)),
    Backoff(MicroSeconds),
    None,
}
```

The above piece of code defines the `Scheduler` trait and the `SchedulerResult` enumeration, which is the result returned by `scheduler()` function. The following structure is the structure that implements the original schedule algorithm:

```rust
struct PriorityScheduler {
    stage_out: Arc<Mutex<Box<[StageOut]>>>,
}
impl SchedulerTrait for PriorityScheduler {
    fn reset(&mut self) -> {}                          // just do nothing
    fn schedule(&mut self) -> ScheduleResult {
        let mut stage_out = zlock!(self.stage_out);
        for (prio, queue) in stage_out.iter_mut().enumerate() {
            match queue.try_pull() {
                Pull::Some(batch) => {
                    return ScheduleResult::Some((batch, prio));
                }
                Pull::Backoff(deadline) => {
                    return ScheduleResult::Backoff(deadline);
                }
                Pull::None => {}
            }
        }
        ScheduleResult::None
    }
}
```

Basically, I just copy the original code into this function. Here, I am going to give another example of scheduler. I implemented a scheduler that combined weighted round robin and priority scheduling. The idea is that we divide the transmission into two phases: contention and polling. In the contention phase, the scheduler will schedule based on priority, while in the polling phases, the scheduler will based on the allocated virtual time.

```rust
struct WRRScheduler {
    stage_out: Arc<Mutex<Box<[StageOut]>>>,
    c_window_start: Option<Instant>,
    n_window_start: Option<Instant>,
    last_update: Option<(Instant, MicroSeconds, usize)>,
    virtual_time: BinaryHeap<Reverse<(MicroSeconds, usize)>>,
    weight: Vec<usize>,
    c_window: MicroSeconds,
    n_window: MicroSeconds,
    is_contention: bool,
    length: usize,
}
```

Some fields store the setting of the scheduler, for instance, `c_window` and `n_window` represent the length of the contention window and the polling window. The `weight` represent the weight of all priorities. Unfortunately, the explanation of the scheduler described above is temporarily unavailable. I am still working on it and hope to resolve this soon.

After having implemented our first customized scheduler, we may also define the enumeration `SchedulerConfig` and the structure `SchedulerBuilder`. The structure `SchedulerBuilder` implements the `build()` method that will return an instance of `Box<dyn SchedulerTrait + Send + Sync>` generics. Precisely speaking, the `build()` method will return the `Scheduler` trait object that is specific by the configuration. [12]

```rust
#[derive(Default)]
#[allow(dead_code)]
enum SchedulerConfig {
    #[default]
    Priority,
    WeightedRoundRobin(WRRConfig),
    Custom(CustomConfig),
}
struct SchedulerBuilder {
    config: SchedulerConfig,
}
```

With those enumeration and structure defined, we can implement the `build()` and other related methods as follow.

```rust
impl SchedulerBuilder {
    fn build(
        self,
        stage_out: Arc<Mutex<Box<[StageOut]>>>,
    ) -> Box<dyn SchedulerTrait + Send + Sync> {
        match &self.config {
            SchedulerConfig::Priority => self.build_priority(stage_out),
            SchedulerConfig::WeightedRoundRobin(config) => {
                self.build_wrr(stage_out, config.clone())
            }
            SchedulerConfig::Custom(config) => {
                self.build_custom(stage_out, config.clone())
            }
        }
    }
    fn build_priority(
        &self,
        stage_out: Arc<Mutex<Box<[StageOut]>>>
    ) -> Box<PriorityScheduler> {
        Box::new(PriorityScheduler { stage_out })
    }
    fn build_wrr(
        &self,
        stage_out: Arc<Mutex<Box<[StageOut]>>>,
        config: WRRConfig,
```

[12] `Send` and `Sync` is added to make compiler not complaining about multi-threading.

```rust
    ) -> Box<WRRScheduler> {
        let length = stage_out.lock().unwrap().len();
        Box::new(WRRScheduler {
            stage_out,
            c_window_start: None,
            n_window_start: None,
            last_update: None,
            virtual_time: BinaryHeap::new(),
            weight: config.weight,
            c_window: config.c_window,
            n_window: config.n_window,
            is_contention: false,
            length,
        })
    }
    /* fn build_custom() is omitted */
}
```

First, the `build()` method will call the corresponding sub-`build` method based on the given configuration. Next, the sub-`build` method will the construct the scheduler structure then returned.