Note on Zenoh

陳信睿
B11902047

November 29, 2024

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

[1] Line 31 in the file commons/zenoh-util/src/log.rs.

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. The `wait()` function attempts to extract the config and passes it to `Session::new(config).wait()`.

The function `Session::new()` [4] 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.

[2] Line 2900 in zenoh/src/api/session.rs
[3] Line 2954 in zenoh/src/api/session.rs
[4] Line 955 in zenoh/src/api/session.rs
[5] Line 126 in zenoh/src/net/runtime/mod.rs

```
# 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()` 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

[6] Implemented at line 70 in zenoh/src/net/routing/router.rs

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:

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)
                // suppose tcp is used
              LinkManagerUnicastTcp::new_link_inner(...)
```

Then the TCP stream socket is returned.

3 On `io/zenoh-transport`

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. 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:

[7] Line 119 in zenoh/src/net/runtime/orchestrator.rs
[8] Line 171 in zenoh/src/net/runtime/orchestrator.rs

```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.

For instance, if all `WBatch` structures in the RingBuffer are unavailable, it could indicate that the production speed exceeds the consumption speed, leading to congestion. In such cases, when a thread requests a `WBatch` but the buffer is unavailable, a backoff mechanism is required to prevent further worsen the congestion. However, I do not fully understand the details of how this mechanism works.

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.

[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.

The following piece of code implements this idea:

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

3.1 The official implementation of the transport statistics structure

In this subsection, I study the file of `io/zenoh-transport/src/common/stats.rs`, and I am going to show how the Zenoh 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 build and run separately */
$ cargo build --example <example_name> --features stats
$ ./target/debug/<example_name>
/* if you are using the tracing crate, you may use the following */
$ RUST_LOG=debug ./target/debug/<example_name>
```

I add `RUST_LOG=debug` environment variables in most cases because instead of using `prinln!()`, I use `tracing::info!()` macro to print out the information (the reason has been mentioned in section 1).

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 and getter method, which is not really inspirable.

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