

Note on Zenoh

陳信睿

B11902047

November 25, 2024

1 Trace Zenoh using ufttrace

The reference provided useful guidance for identifying the correct compile flag. To trace the Zenoh program using ufttrace, follow these steps:

```
1 /* modify the toolchain */
2 $ cd $zenoh-repo/
3 $ vim rust-toolchain.toml // modify to "nightly"

5 /* install ufttrace and compile zenoh */
6 $ sudo apt install ufttrace
7 /* build zenoh using the following command lines */
8 $ export RUSTFLAGS=\ // Leading spaces are not allowed in the next line
9 "-Z instrument-mcount -C passes=ee-instrument<post-inline>"
10 $ cargo build --all-targets
11 $ ufttrace record ./target/examples/z_pub
12 $ ufttrace {replay,graph,tui}

14 /* To enable the rust tracing::debug!() macro,
15 use the following to run the program */
16 $ RUST_LOG=debug cargo run --example z_pub
```

2 Case study: The z_pub example

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

The function `zenoh::open(config)`¹ returns an `OpenBuilder<TryIntoConfig>` instance, a structure that implements the `IntoFuture` trait. Therefore, the `.await` operator asynchronously waits for the `wait()`² function to complete. The `wait()` function attempts to extract the config and passes it to `Session::new(config).wait()`.

¹ Line 2900 in `zenoh/src/api/session.rs`

² Line 2954 in `zenoh/src/api/session.rs`

The function `Session::new()`³ first collects the topics it subscribes to and publishes. It then initializes a new runtime by

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

Let's dive into the details of runtime and `RuntimeBuilder::build()`⁴. 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:

```
0 # RuntimeBuilder::build() in zenoh/src/net/runtime/mod.rs
1 let router = Arc::new(Router::new(zid, whatami, hlc.clone(), &config)?);
143 let tm_builder = TransportManager::builder()
149     .from_config(&config)
150     .await?
151     .whatami(whatami)
152     .zid(zid);
159 let transport_manager = tm_builder.build(handler.clone())?;
169 let config = Notifier::new(config);
170 let runtime = Runtime {
171     state: Arc::new(RuntimeState {
185         ... // ommited
186     }),
187 };
```

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.

```
0 # Session::new() in zenoh/src/api/session.rs
971 let mut session = Self::init(
972     runtime.clone(),
973     aggregated_subscribers,
974     aggregated_publishers,
975 )
976 .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()`

³ Line 955 in `zenoh/src/api/session.rs`

⁴ Line 126 in `zenoh/src/net/runtime/mod.rs`

function is invoked, which subsequently calls `router.new_primitives()`⁵. 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:

```
0 # Router::new_primitive() in zenoh/src/net/routing/router.rs
78 let fid = tables.face_counter;
79 tables.face_counter += 1;
80 let newface = tables
81     .faces
82     .entry(fid)
83     .or_insert_with(|| {
84         FaceState::new(
85             fid,
86             zid,
87             WhatAmI::Client,
88             #[cfg(feature = "stats")]
89             None,
90             primitives.clone(),
91             None,
92             None,
93             ctrl_lock.new_face(),
94         )
95     })
96     .clone();
97 tracing::debug!("New {}", newface);
98 for (key, val) in tables.faces.iter() {
99     tracing::debug!("key: {key} val: {val}");
100 }
```

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 `fids` 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 invoked⁶. 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()` method⁷. In the `Runtime` method `Runtime::start_peer()`, the following operations are performed:

⁵ Implemented at line 70 in `zenoh/src/net/routing/router.rs`

⁶ Line 119 in `zenoh/src/net/runtime/orchestrator.rs`

⁷ 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:

```

1 Runtime::bind_listeners(&listeners)
2   Runtime::bind_listeners_impl(listeners)
3     Runtime::add_listener(&self, listener: EndPoint)
4       Runtime::manager().add_listener(endpoint)
5         TransportManager::add_listener_unicast(endpoint)
6           TransportManager::new_link_manager_unicast(...)
7             LinkManagerUcastTcp::new_listener(endpoint)
              // suppose tcp is used
8             LinkManagerUcastTcp::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 io/zenoh-transport, 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 (**M**ultiple **P**roducer, **S**ingle **C**onsumer) 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:

```

1 pub struct WBatch {
2     pub buffer: BBuf,           // The buffer to perform the batching on
3     pub codec: Zenoh080Batch,   // The batch codec
4     /* It contains 1 byte to signal whether the batch is compressed */
5     pub config: BatchConfig,
6     #[cfg(feature = "stats")]    // Statistics related to this batch
7     pub stats: WBatchStats,
8 }

```

Essentially, the WBatch structure 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.⁸

(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:

```
1 macro_rules! zgetbatch_rets {
2   ($fragment:expr, $restore_sn:expr) => {
3     loop {
4       match c_guard.take() {
5         Some(batch) => break batch,
6         None => match self.s_ref.pull() {
7           Some(mut batch) => {
8             batch.clear();
9             self.s_out.atomic_backoff.first_write.store(
10               LOCAL_EPOCH.elapsed().as_micros() as MicroSeconds,
```

⁸ 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.

```

11         Ordering::Relaxed);
12         break batch;
13     }
14     None => {
15         drop(c_guard);
16         match deadline_before_drop {
17             Some(deadline) if !$fragment => {
18                 if !self.s_ref.wait_deadline(deadline) {
19                     $restore_sn;
20                     return false
21                 }
22             }
23             _ => {
24                 if !self.s_ref.wait() {
25                     $restore_sn;
26                     return false;
27                 }
28             }
29         }
30         c_guard = self.mutex.current();
31     }
32 },
33 }
34 }
35 };
36 }

```

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.

```

1  /* StageIn */
2  struct StageInRefill {
3      n_ref_r: Waiter,
4      s_ref_r: RingBufferReader<WBatch, RBLLEN>,
5  }
6  struct StageInOut {
7      n_out_w: Notifier,
8      s_out_w: RingBufferWriter<WBatch, RBLLEN>,
9      atomic_backoff: Arc<AtomicBackoff>,
10 }
11 struct StageInMutex {
12     current: Arc<Mutex<Option<WBatch>>>>,
13     priority: TransportPriorityTx,
14 }
15 /* StageOut */
16 struct StageOutIn {
17     s_out_r: RingBufferReader<WBatch, RBLLEN>,
18     current: Arc<Mutex<Option<WBatch>>>>,
19     backoff: Backoff,
20 }
21 struct StageOutRefill {
22     n_ref_w: Notifier,
23     s_ref_w: RingBufferWriter<WBatch, RBLLEN>,
24 }

```

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 `StageInOut` structure stores the information required to notify the consumer threads.