Lesson 8: Using Applications and Endpoints

In this lesson we’ll begin building a more general-purpose FTL publisher and subscriber. Ideally, when you’re finished, you’ll have a set of good reference samples for use as a starting point for your own FTL client programs.

But first, let’s look at what the new client programs do.

8.1 The Client Programs

For our samples, the publisher and subscriber implement an application-level protocol consisting of two types of messages: control and data. Each message contains a subset of seven fields, listed in Table 8.1.

Each set of data messages is a stream. We use beginning of stream (BOS) and end of stream (EOS) to indicate where the stream starts and ends.

Sequence numbers denote where a message belongs in a stream.

8.1.1 Publisher Operation

The publisher begins by sending a control message. Included in this message are four fields: "tag" (containing "control"), "count", "contents", and "bos". "count" contains the number of data messages the subscriber should expect to receive. This message marks the beginning of the stream of messages being published.

Next, the publisher sends zero or more data messages. Each message contains four fields: "tag" (containing "data"), "contents", "seq", and "even". "seq" contains the data message sequence number, counting from 1. "even" is set to 1 if "seq" is even, and 0 if it is odd.

When all data messages have been sent, the publisher sends a final control message. This message contains four fields: "tag" (containing "control"), "count", "contents", and "eos". "count" contains the number of data messages the publisher sent. This message marks the end of the stream of messages being published.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;tag&quot;</td>
<td>String</td>
<td>Indicates the message type (&quot;control&quot; or &quot;data&quot;).</td>
</tr>
<tr>
<td>&quot;count&quot;</td>
<td>Long</td>
<td>Contains the number of data messages to expect.</td>
</tr>
<tr>
<td>&quot;contents&quot;</td>
<td>String</td>
<td>Payload field containing some text.</td>
</tr>
<tr>
<td>&quot;seq&quot;</td>
<td>Long</td>
<td>Data message sequence number.</td>
</tr>
<tr>
<td>&quot;bos&quot;</td>
<td>Long</td>
<td>Message is the beginning of stream (BOS) if present.</td>
</tr>
<tr>
<td>&quot;eos&quot;</td>
<td>Long</td>
<td>Message is the end of stream (EOS) if present.</td>
</tr>
<tr>
<td>&quot;even&quot;</td>
<td>Long</td>
<td>Contains 1 if the &quot;seq&quot; field is even, and 0 if odd.</td>
</tr>
</tbody>
</table>

Table 8.1: Message fields
8.1.2 Subscriber Operation

The subscriber will expect messages in the order listed above: a control message indicating BOS, zero or more data messages, followed by a control message indicating EOS. Any deviation from this order should be detected by the subscriber and cause a message to be printed.

8.2 The Code

The code files for this lesson can be found in the $TIBFTL_TUTDIR/code/lesson-08 directory. Four files are involved: Makefile, common.h, ftlpub.c, and ftlsub.c. Instead of listing the entire contents of each file, we will summarize the interesting parts of each.

8.2.1 common.h

Listing 8.1: common.h

Most of common.h is unchanged, but we’ve added definitions for the message fields.

8.2.2 ftlpub.c

Listing 8.2: ftlpub.c Part 1

To make the applications more generally useful, we will be moving away from hard-coding values directly in the code, and instead allow values to be specified on the command line. A variable is provided for each such value, and reasonable default values established via #define statements. This approach allows us to change the values for different invocations of a program without having to modify the source code and rebuild the program.

Listing 8.2 shows the variables (and default values) for the FTL server URL, application name, endpoint name, and number of messages to send.
const char * Options = "a:c:e:hr:"

void print_usage(void)
{
    fprintf(stderr,
        "Usage: ftlpub [options]\n"
        " -a APPLICATION set application name\n"
        " -c COUNT send COUNT data messages\n"
        " -e ENDPOINT set endpoint name\n"
        " -h print this help\n"
        " -r URL use realm service URL\n"
    );
    exit(1);
}

Listing 8.3: ftlpub.c Part 2

The standard getopt() function is used to process command-line arguments. The application (-a), endpoint (-e), message count (-c), and FTL server (-r) can be specified on the command line.

Listing 8.4: ftlpub.c Part 3

The beginning of main() hasn’t changed much. We start by parsing any arguments given on the command line.

The remainder of Listing 8.4 is similar to what we have seen before. However, now we’re using a variable for the FTL server rather than a string constant. Instead of passing NULL, variables are used for the application and endpoint names.
printf("using Realm Service=%s\n\tApplication=%s\n\tEndpoint=%s\n",
    RealmService,
    (Application != NULL ? Application : "default"),
    (Endpoint != NULL ? Endpoint : "default"));
printf("sending initial message\n");
fflush(stdout);
tibMessage_SetString(ex, msg, FIELD_NAME_TAG, FIELD_TAG_CONTROL);
tibMessage_SetString(ex, msg, FIELD_NAME_CONTENTS, "Initial message");
tibMessage_SetLong(ex, msg, FIELD_NAME_COUNT, MsgCount);
tibMessage_SetLong(ex, msg, FIELD_NAME_BOS, 1);
tibPublisher_Send(ex, pub, msg);
CHECK(ex);

Listing 8.5: ftlpub.c Part 4

In order to confirm the command-line options specified, the program prints the values used for the FTL server URL, application, and endpoint. Next it builds the initial control message, which contains four fields as described in Section 8.1. The initial control message is sent and the exception object is checked.

tibMessage_SetString(ex, msg, FIELD_NAME_TAG, FIELD_TAG_DATA);
tibMessage_ClearField(ex, msg, FIELD_NAME_COUNT);
tibMessage_ClearField(ex, msg, FIELD_NAME_BOS);
tibMessage_SetString(ex, msg, FIELD_NAME_CONTENTS, "Data message");

Listing 8.6: ftlpub.c Part 5

The next step is to set the fields required for the data messages. The "tag" field is set to the required value using the symbols defined for the purpose rather than hard-coded strings.

Notice that we didn’t have to create a new message; we are using the message that was originally created to send the initial control message. This brings up an important concept:

**Concept 6. A message object can be reused multiple times.**

The "contents" field is set to a new value. You may recall that this field was set while building the initial control message. This highlights another important concept about FTL messages:

**Concept 7. Setting a field value overwrites any existing value within the message.**

Because data messages do not contain the "count" or "bos" fields, those fields are explicitly cleared. Another important concept:

**Concept 8. Clearing an existing field in a message sets its value to empty. When sending a message, FTL does not send empty fields.**

Even though the "count" and "bos" fields exist in the message (since we explicitly set them earlier), clearing them marks those fields as empty, effectively removing them from the message.
for (idx = 1; idx <= MsgCount; ++idx)
{
    tibMessage_SetLong(ex, msg, FIELD_NAME_SEQ, idx);
    if ((idx % 2) == 0)
    {
        tibMessage_SetLong(ex, msg, FIELD_NAME_EVEN, 1);
    }
    else
    {
        tibMessage_SetLong(ex, msg, FIELD_NAME_EVEN, 0);
    }
    tibPublisher_Send(ex, pub, msg);
    CHECK(ex);
    sleep(1);
}

Listing 8.7: ftlpub.c Part 6

Listing 8.7 contains the code to build and send the data messages. First, the "seq" field is set to the message number. Next the program sets the "even" field; to 1 if the message number is even, and to 0 if it is odd. This is not too useful now, but will be in upcoming lessons. Then the message is sent and the exception checked.

Notice that the loop code modifies only the "seq" and "even" fields of the message. The "tag" and "contents" fields remain the same in every data message.

**Concept 9.** When a message object is reused, fields that were previously set remain set to those values.

And one more important concept:

**Concept 10.** When reusing a message object, clear any fields which are no longer relevant.

Which is why the "count" and "bos" fields are cleared in Listing 8.6.

Note the sleep(1) call in line 121. This is purely for convenience, so that the publisher will run slow enough for us to easily see what it is doing. It is not recommended that your client programs do this — in almost every case, you want things to run as fast as possible and send messages as soon as they are ready to be sent.

printf("sending final message\n");
fflush(stdout);
tibMessage_ClearAllFields(ex, msg);
tibMessage_SetString(ex, msg, FIELD_NAME_TAG, FIELD_TAG_CONTROL);
tibMessage_SetString(ex, msg, FIELD_NAME_CONTENTS, "Final message");
tibMessage_SetLong(ex, msg, FIELD_NAME_COUNT, MsgCount);
tibMessage_SetLong(ex, msg, FIELD_NAME_EOS, 1);
tibPublisher_Send(ex, pub, msg);
CHECK(ex);

Listing 8.8: ftlpub.c Part 7

Next we build and send the final control message. Rather than clearing each individual field (using tibMessage_ClearField()), we use tibMessage_ClearAllFields() to clear all fields in the message.
Finally, any FTL objects we created are destroyed, in the reverse order of their creation.

8.2.3 ftlsub.c

In a manner similar to that in the publisher, our latest version moves away from hard-coded values.

We declare variables for the FTL server, application name, endpoint name, and matcher string, each initialized to their default values. We also declare a `Verbose` variable to allow us to control the amount of output at runtime.

In Listing 8.11 we define variables to support our previously defined application-level protocol. `Done` is used to indicate when all messages have been received; counters are defined to keep track of the number of data and control messages received; and `ExpectedMsgCount` stores the number of data messages we expect to receive (from the initial control message).
void onMessages(tibEx ex, tibEventQueue msgQueue, tibint32_t msgNum, tibMessage *msgs,
    void **closures)
{
    tibint32_t i;
    char buffer[1024];
    static tibbool_t init_received = tibfalse;
    const char * type = NULL;

    for (i = 0; i < msgNum; i++)
    {
        if (Verbose)
        {
            printf("received message\n");
            (void)tibMessage_ToString(ex, msgs[i], buffer, sizeof(buffer));
            printf(" %s\n", buffer);
            fflush(stdout);
        }
        if (!tibMessage_IsFieldSet(ex, msgs[i], FIELD_NAME_TAG))
        {
            printf("%s field is not present in received message\n", FIELD_NAME_TAG);
            fflush(stdout);
            continue;
        }
        type = tibMessage_GetString(ex, msgs[i], FIELD_NAME_TAG);
    }
}

Listing 8.12: ftlsub.c Part 3

The `onMessages()` function is a little more sophisticated than the one in `ftlbasicsub.c` used for the first six lessons, although the basic structure is the same. `init_received` is used to keep track of whether the first control message has been received (refer back to Section 8.1 for a description of the protocol).

The `Verbose` variable controls whether or not the full message is printed.

Before fetching the value of a field, it is good practice to ensure that the field exists. We do this before retrieving the value of the "tag" field from the message.
if (strcmp(type, FIELD_TAG_CONTROL) == 0)
{
    ControlMsgsReceived++;
    if (init_received)
    {
        /* Looking for eos */
        if (tibMessage_IsFieldSet(ex, msgs[i], FIELD_NAME_EOS))
            {
                printf("received EOS indicator\n");
                fflush(stdout);
                Done = tibtrue;
            } else
            {
                printf("received unexpected control message\n");
                fflush(stdout);
            }
    } else
    {
        /* looking for bos */
        if (tibMessage_IsFieldSet(ex, msgs[i], FIELD_NAME_BOS))
            {
                init_received = tibtrue;
                printf("received BOS indicator\n");
                fflush(stdout);
                ExpectedMsgCount = tibMessage_GetLongLong(ex, msgs[i], FIELD_NAME_COUNT);
            } else
            {
                printf("received unexpected control message\n");
                fflush(stdout);
            }
    }
}

Listing 8.13: ftlsub.c Part 4

Control messages are handled in Listing 8.13. First the counter keeping track of the number of control messages received is updated. Next we check to see if the initialization control message (BOS) has already been received.

If so, the only control message we expect to see is the EOS message, so we check to see if the "eos" field is present. If it is present, print a message indicating so and set the Done flag to tibtrue. Otherwise print a message indicating we received an unexpected control message.

If init_received is not tibtrue, indicating we have not yet received the BOS control message, then check for the presence of the "bos" field. If the field is present, set the init_received flag to indicate a BOS has been received, print a message indicating so, and fetch the "count" field, which indicates how many data messages are expected.

If a BOS has not been received, and the message does not contain the "bos" field, then we have received an unexpected control message; print a message indicating this has occurred.
else if (strcmp(type, FIELD_TAG_DATA) == 0)
{
  DataMsgsReceived++;
  if (init_received)
  {
    if (Verbose)
    {
      if (tibMessage_IsFieldSet(ex, msgs[i], FIELD_NAME_SEQ))
      {
        tibint64_t seq = tibMessage_GetLong(ex, msgs[i], FIELD_NAME_SEQ);
        printf("received data seq %" PRId64 "\n", seq);
      }
      else
      {
        printf("%s field not present in received message\n", FIELD_NAME_SEQ);
      }
      fflush(stdout);
    }
    else
    {
      printf("received unexpected data message\n");
      fflush(stdout);
    }
  }
  else
  {
    printf("received unexpected message\n");
    fflush(stdout);
  }
}

Listing 8.14: ftlsub.c Part 5

Listing 8.14 handles the receipt of data messages. First the counter keeping track of the number of data messages received is updated. Based on how the protocol is designed, once we’ve received a BOS message we expect a series of data messages.

For a data message we extract and print the sequence number field, but only if the field exists and verbose output is enabled.

If a BOS message has not been previously received, then we have received an unexpected data message; print a message indicating this.

else
{
  printf("received unexpected message\n");
  fflush(stdout);
}

Listing 8.15: ftlsub.c Part 6

Listing 8.15 handles the situation in which the "tag" field contained an invalid value.
Listing 8.16: ftlsub.c Part 7

`main()` accesses and parses the command line arguments as well as declaring some of the basic FTL objects needed in the same manner as earlier sample code.

An FTL exception object is created, but now the call to `tibRealm_Connect()` passes a variable for the FTL server instead of a string constant, as well as passing a variable for the application name instead of `NULL` (forcing the use of the default application).

The calls to create a matcher, subscriber, and event queue should look familiar, except that a variable is passed to `tibSubscriber_Create()` for the endpoint name instead of `NULL`.

Notice that the matcher, if created, is destroyed immediately after the call to `tibSubscriber_Create()`. It is good practice to destroy objects when they are no longer needed.
printf("using Realm Service=%s
\tApplication=%s
\tEndpoint=%s
\tMatcher=%s\n",
RealmService,
(Application != NULL ? Application : "default"),
(Endpoint != NULL ? Endpoint : "default"),
(Matcher != NULL ? Matcher: "none"));
printf("waiting for message(s)\n");
fflush(stdout);
do
{
  tibEventQueue_Dispatch(ex, queue, TIB_TIMEOUT_WAIT_FOREVER);
}while (!Done && tibEx_GetErrorCode(ex) == TIB_OK);
printf("Received %" PRId64 " control, and %" PRId64 " of %" PRId64 " data messages.\n",
ControlMsgsReceived, DataMsgsReceived, ExpectedMsgCount);
fflush(stdout);
tibEventQueue_RemoveSubscriber(ex, queue, sub, NULL);
tibEventQueue_Destroy(ex, queue, NULL);
tibSubscriber_Close(ex, sub);
tibRealm_Close(ex, realm);
tib_Close(ex);
CHECK(ex);
tibEx_Destroy(ex);

Listing 8.17: ftlsub.c Part 8

As in ftlpub.c, the command-line option values being used are printed. You’ll also recognize the dispatch loop as it has not changed from our earlier samples. After all messages have been received from the publisher, the number of control and data messages received are printed. Finally, the usual cleanup actions are performed.

8.3 Build and Run the Sample Applications

To build and run the sample applications, make sure the FTL server is running and open a command window. Assuming you have set the TIBFTL_DIR environment variable, navigate to $TIBFTL_TUTDIR/code/lesson-08, build the samples, and start the subscriber:

make
./ftlsub

In another command window, navigate to $TIBFTL_TUTDIR/code/lesson-08 and start the publisher:

./ftlpub

You should see the following output from the publisher:

using Realm Service=http://localhost:8080
Application=default
Endpoint=default
sending initial message
sending 20 data messages
sending final message

Listing 8.18: ftlpub output

As expected, ftlpub sent the initial control message (BOS), sent 20 data messages (the default number), and send the final control message (EOS).
The output in the subscriber command window should appear as:

```java
using Realm Service=http://localhost:8080
Application=default
Endpoint=default
Matcher=none
waiting for message(s)
received BOS indicator
received EOS indicator
Received 2 control, and 20 of 20 data messages.
```

Listing 8.19: ftlsub output

Both programs appear to be operating correctly.

Now try running the subscriber with the `-v` option. In the subscriber command window, restart the subscriber:

```
./ftlsub -v
```

In the publisher command window, restart the publisher but only send five messages:

```
./ftlpub -c 5
```

The output from `ftlpub` is the same as the first time, but the output from `ftlsub` includes more information:

```java
using Realm Service=http://localhost:8080
Application=default
Endpoint=default
Matcher=none
waiting for message(s)
received message
   {string:tag="control", string:contents="Initial message", long:count=5, long:bos=1}
received BOS indicator
received message
   {string:tag="data", string:contents="Data message", long:seq=1, long:even=0}
received data seq 1
received message
   {string:tag="data", string:contents="Data message", long:seq=2, long:even=1}
received data seq 2
received message
   {string:tag="data", string:contents="Data message", long:seq=3, long:even=0}
received data seq 3
received message
   {string:tag="data", string:contents="Data message", long:seq=4, long:even=1}
received data seq 4
received message
   {string:tag="data", string:contents="Data message", long:seq=5, long:even=0}
received data seq 5
received message
   {string:tag="control", string:contents="Final message", long:count=5, long:eos=1}
received EOS indicator
Received 2 control, and 5 of 5 data messages.
```

Listing 8.20: ftlsub -v output

As a final test, let’s run the applications with different application and endpoint names. Specifically, we’ll use the “App1” application and “endpoint-1” endpoint we created in lesson 7 for both `ftlpub` and `ftlsub`.
In the subscriber command window, start the subscriber with the options needed to set the application and endpoint:

```
./ftlsub -a App1 -e endpoint-1
```

In the publisher command window, start the publisher with the same options:

```
./ftlpub -a App1 -e endpoint-1
```

The resulting output from the publisher reflects the changed application and endpoint names:

```
using Realm Service=http://localhost:8080
   Application=App1
   Endpoint=endpoint-1
sending initial message
sending 20 data messages
sending final message
```

Listing 8.21: `ftlpub -a App1 -e endpoint-1` output

As does the subscriber output:

```
using Realm Service=http://localhost:8080
   Application=App1
   Endpoint=endpoint-1
   Matcher=none
waiting for message(s)
received BOS indicator
received EOS indicator
Received 2 control, and 20 of 20 data messages.
```

Listing 8.22: `ftlsub -a App1 -e endpoint-1` output

### 8.4 Order Matters

Up to this point, we’ve always run the subscriber application first, and the publisher application second. This time we’ll run the publisher first.

In the publisher command window start the publisher:

```
./ftlpub
```

The resulting output is:

```
using Realm Service=http://localhost:8080
   Application=default
   Endpoint=default
sending initial message
sending 20 data messages
sending final message
```

Listing 8.23: `ftlpub` output
The publisher reports that it sent all messages successfully, but we haven’t run the subscriber yet. Let’s start the subscriber and see what it receives. In the subscriber command window:

```
./ftlsub
```

The resulting output is familiar up to a point:

```
using Realm Service=http://localhost:8080
Application=default
Endpoint=default
Matcher=none
waiting for message(s)
```

**Listing 8.24: ftlsub output**

It seems as if the subscriber is waiting for messages but not receiving any. After all, the messages it was expecting to receive have already been sent by the publisher. This is an important idea in FTL messaging systems:

**Concept 11.** At the application level, publishers and subscribers are completely independent (loosely coupled). Publishers are unaware of the presence (or absence, or even the number) of subscribers.

So a publisher will send messages regardless of whether or not subscribers exist; more precisely, a send can succeed whether or not subscribers exist. Obviously, subscribers are indirectly aware of the presence or absence of publishers by virtue of messages being received (or not received). Subscribers can subscribe to endpoints even when no publisher is sending on the corresponding endpoint.

It is important to note that this independence is at the application level. No claims are made regarding independence at the transport level.

The independence of publishers and subscribers does not mean that the model of publisher/subscriber awareness is invalid. There may be situations in which the approach of publishers only sending messages if subscribers are available to consume them is a valid problem approach. However, special application logic is required to synchronize publishers and subscribers if this type of behavior is required.

Make sure you stop the subscriber via `ctrl + C` before continuing to the next section.

### 8.5 Clients

To get this part of the lesson started, start the subscriber using application “App1” and the proper endpoint:

```
./ftlsub -a App1 -e endpoint-1
```

Restart the publisher using application “App1” and the proper endpoint; but for this part of the lesson we need the publisher to run for a much longer period of time, so tell it to send 1000 messages:

```
./ftlpub -c 1000 -a App1 -e endpoint-1
```

Open a web browser and navigate to the URL http://localhost:8080. If the FTL server login page is displayed, login as you did in Section 6.1, page 27. Bring up the Applications grid by clicking on the link on the left side of the screen.
Figure 8.1: Applications grid

Note the changes in the top line of the display:

![Applications grid](image)

Figure 8.2: Status Detail

In previous lessons, the number above “Clients” was 0. Now we have started two client programs, `ftlsub` and `ftlpub`. The information in this display indicates the actual number of active client programs.

Clicking anywhere in this area of the screen will display the Status Dashboard. The dashboard slides down to reveal a status table which gives more specific counts on running client programs and services as shown in Figure 8.3.
The dashboard shows there are currently two clients running. All categories to the right of “Clients” are considered *FTL Services* and are included in the count of services shown at the top of the screen.

Each category can be clicked on to display information about the known FTL programs in that category. Click “Clients” in the dashboard to display client information, as shown in Figure 8.4.

It is important to note that the Status Dashboard and the more detailed status displays are not a separate page: they appear over any page in the user interface without affecting the page underneath. Status can be shown at any time and at any place in the realm service web interface without impacting the information displayed underneath.
A row of summary information is displayed for each client. The information shown for each client includes:

- "Status", representing the current status of the client (Running, Timed Out, etc.).
- "Client Label", which will be discussed in lesson 9.
- "ID", a unique ID assigned to each client by the realm service.
- "Host", the machine on which the client is running.
- "Application", the application name used by the client.
- "Instance", the application instance used by the client. Instances are discussed in a later lesson.
- "Core Server", the realm service to which the client application initially connected.

At the right end of each row are three icons which allow control of the client program:

- Change the client’s logging level
- Change the client’s monitoring mode
- Disable the client

Clicking anywhere in the first client row will show additional information about that specific client:
Figure 8.5: Client Detail

Details about the client are displayed in a scrolling area. Each section is initially hidden but can be revealed by clicking the icon to the left of the section title. On the right side of the client information is a series of shortcuts to specific client details, organized into functional groups: Figure 8.5 shows “Client Information”, “Application”, “Endpoint”, “Queues”, “Recent Advisories”, “Recent Logs”, and “Transport”. You can scroll through the various sections, or use the shortcuts to jump directly to a specific section.

Click the icon to the left of “Client Information” section to display additional details related to this client:
In later lessons, we will cover the various metrics displayed, along with monitoring in general.

### 8.6 Identifying Clients

Looking back at Figure 8.5, page 62, which client program is being displayed? We were running both a publisher and a subscriber, but which one is this? Referring back to Figure 8.4, page 61, which of the two clients is the publisher and which is the subscriber?

Unfortunately, there is no readily apparent way to determine which client is the publisher and which is the subscriber.

The recommended way to identify clients is through the use of properties, which is covered in the next lesson.