Assignment

Reactive Followers

To get started, download the reactive-followers.zip handout archive file and extract it somewhere on your machine.

This will be the final exercise of this course, so hang in there and give it your best!

Acknowledgements: This task is a simplified version of the "Follower Maze" Developer Challenge devised by Amar Shah of SoundCloud. You can see the original task here: follower-maze instructions. Please make sure to follow instructions for this course, and not the follower-maze directly though!

The Challenge

The challenge proposed here is to build a system which acts as a socket server, reading events from an **event source** and forwarding them when appropriate to user **clients**.

There will be two types of clients connecting to your server:

- **One event source**: It will send you a stream of events which *may or may not* require clients to be notified,
- **Many user clients**: Each one representing a specific user, these wait for notifications for events which would be relevant to the user they represent.

After the client identification is sent, the user client starts waiting for events to be sent to them. Events coming from event source should be sent to relevant user clients exactly like read, no modification is required or allowed.

The events model actions such as “a user follows another user”, “a user sends a private message to another user”, “a user updates its status”, etc. Each event is associated to a specific delivery semantics (eg. a private message should only be sent to the target user).

Most of the message parsing is implemented for you already, so you can focus on putting the processing together, without having to focus on the parsing itself.

Packages layout

The `followers.model` package provides case classes modeling the different kinds of messages the server will send and receive.

The `followers.model.Event` data type models the types of events handled by the server.

The `followers.model.Identity` case class models the message sent by a user client to identify itself to the server.

You will have to implement the server in the `followers.Server` class. The task has been broken down into smaller pieces so that you can just walk through this file and implement these pieces one after the other.

The handout also includes tests (in the `src/test/scala` directory) that you can use to check that your implementation is correct. Note that our graders will run more tests than those provided in the handout. Feel free to add your own tests to the suite.

Framing and parsing events

On the inbound (receiving events) side of the server you have to take into account that the incoming data **may or may not** be framed exactly as "1 byte string containing exactly one event".

After all, a transport protocols such as TCP is streaming protocol that only guarantees that bytes will be delivered, but along the way from clients to servers, the payloads could be (and often are) chunked up by intermediaries. In other words, even if the event source sends an exact `Follow` event, as exactly the ByteString: `666|F|60|50\n` the server may still receive it as a first chunk containing `666|F|60` followed by another one containing `|50\n`. The same applies for multiple events; i.e. if a client sends two messages to follow different ids, such as: `666|F|60|50\n` and then `667|F|60|45\n`, the server may receive them "lumped together" into one ByteString (!).

In order to solve this, we will need to apply some `Framing` strategy. Thankfully the protocol is rather simple, and messages are always separated with `\n` characters, so a simple delimiter based framing will be enough here.

Implement the `reframedFlow` member: a `Flow` that turns incoming chunks of bytes into `String` values containing exactly one message.

Then, use `reframedFlow` to implement `eventParserFlow`: a flow producing high-level `Event` values from the incoming chunks of bytes.

Finally, implement `identityParserSink`: a `Sink` that consumes chunks of bytes, takes the first message from it and parses it as an `Identity` value. This `Sink` will eventually materialize the parsed `Identity` value.

Dealing with late arrivals

Part of the task is to deal with late arrivals of data into the stream.

There is a number of ways one can solve this, though all of them will require some form of buffering. In our exercise the task is somewhat simplified since we are dealing with one stream, and can (for simplification) deal with the entire incoming stream by reintroducing ordering into it.

Note: In real systems this topic is often expanded into dealing with multiple independent streams of events arriving at the same endpoint, so such reordering stage has to know about their identity and reintroduce ordering for each of the streams -- this often is referred to as a "late arrival groupBy".

Implement the `reintroduceOrdering` member: a `Flow` that consumes unordered events and produces events ordered by their `sequenceNr` value.

Followers Map

The next step consists in keeping track of who follows who, according to the `Follow` and `Unfollow` received events.

Implement the `followersFlow` member: a `Flow` that consumes events and produces the same events paired with the state of followers at the point each event has occurred.

For instance, if we start from a state where nobody follows nobody, and we receive an `Event.Follow(1, fromUserId = 2, toUserId = 3)`, then the state of followers should say that user 2 follows user 3: `Map(2 -> Set(3))`.

Once you are able to compute the map of followers associated with each event, you can implement in the `isNotified` method the logic that defines whether an event should be delivered to a given user or not.

Hub

In the last part of this task you have to implement a streaming engine that supports dynamic connections: an arbitrary number of clients should be able to connect to the server and get their custom notifications feed.

To achieve that the server uses a `BroadcastHub`: a graph stage that materializes to a `Source`, which, itself, can be materialized by as many clients as we want. The `BroadcastHub` consumes the events and forwards them to all the clients consuming the materialized `Source`.

Similarly, the server should allow the event source to be connected dynamically (after the server has started). That's why it uses a `MergeHub` for it.

The event source is processed by the `incomingDataFlow` flow, which re-frames the byte chunks, decodes the events, re-orders them, and associates the state of followers (who follows who) to each event.

Then, `eventsFlow` provides a flow that can be used to connect a source of event to the server. The server does not "reply" to the event source, that's why the output type of the `Flow` is `Nothing`.

The `outgoingFlow` method is an intermediate step for implementing the user connection endpoint. It defines a source of events to be delivered to a given user.

Finally, the `clientFlow` method defines the user connection endpoint. It is a flow that receives the user identity and then produces the feed of events for this user.

Submission

10 points possible (graded)
Run the packageSubmission command and upload the submission.jar file here.

Choose Files No file chosen

Submit