Streaming gRPC

In the **first example (index.html)** we saw a gRPC service call for single request reply. Let's add a bidirectional streaming call. First we will run it and then look at how it's implemented.

Running the streaming call

To run Hello World including the streaming calls:

1. Run the server:

sbt | Maven | Gradle

```
mvn compile dependency:properties exec:exec@server
```

Maven builds the project and runs the gRPC server

This starts the server in the same way as in the first example we ran previously. The output should include something like:

```
gRPC server bound to: /127.0.0.1:8080
```

2. Run the client, open another console window and enter:

sbt | Maven | Gradle

```
mvn -DGreeterClient.user=Alice compile dependency:properties exec:e
```

Maven runs the gRPC client for Alice

Note that the difference from the first example is the additional argument `-DGreeterClient.user=Alice`.

The output should include something like:

```
Performing request: Alice
Performing streaming requests: Alice
HelloReply(Hello, Alice)
Alice got streaming reply: Hello, Alice-0
Alice got streaming reply: Hello, Alice-1
Alice got streaming reply: Hello, Alice-2
Alice got streaming reply: Hello, Alice-3
```

The “Performing request: Alice” and “HelloReply(Hello, Alice)” comes from the single request response call in the previous example and the “streaming” are new.

3. Open yet another console window and enter:

sbt | **Maven** | **Gradle**

```
mvn -DGreeterClient.user=Bob compile dependency:properties exec:exe
```

Maven runs the gRPC client for Bob

Note that the difference is the argument `Bob`. The output should include something like:

```
Performing request: Bob
Performing streaming requests: Bob
HelloReply(Hello, Bob)
Bob got streaming reply: Hello, Bob-0
Bob got streaming reply: Hello, Alice-38
Bob got streaming reply: Hello, Bob-1
Bob got streaming reply: Hello, Alice-39
Bob got streaming reply: Hello, Bob-2
Bob got streaming reply: Hello, Alice-40
Bob got streaming reply: Hello, Bob-3
```

Note how the messages from Alice are also received by Bob.

4. Switch back to the console window with the Alice client. The output should include something like:

```
Alice got streaming reply: Hello, Bob-10
Alice got streaming reply: Hello, Alice-48
Alice got streaming reply: Hello, Bob-11
Alice got streaming reply: Hello, Alice-49
Alice got streaming reply: Hello, Bob-12
Alice got streaming reply: Hello, Alice-50
Alice got streaming reply: Hello, Bob-13
```

Note how messages from both Alice and Bob are received in both clients. The streaming request messages are broadcasted to all connected clients via the server.

Now take a look at how this is implemented.

You can end the programs with `ctrl-c`.

What the streaming Hello World does

As you saw in the console output, the example outputs greetings from all clients to all clients. Let's take at the code and what happens at runtime.

Server

First, the `GreeterServer` main class is the same as explained in the **first example (index.html#server)**. It binds the `GreeterServiceImpl` to the HTTP server.

We define the interface of the the new call in the protobuf file `src/main/proto/helloworld.proto` next to the previous `SayHello` call:

```
copy// The stream of incoming HelloRequest messages are
// sent out as corresponding HelloReply. From
// all clients to all clients, like a chat room.
rpc SayHelloToAll (stream HelloRequest) returns (stream HelloReply) {}
```

This method is generated in the `GreeterService` interface and we have to implement it on the server side in `GreeterServiceImpl`:

copy

```
import akka.NotUsed;
import akka.japi.Pair;
import akka.actor.typed.ActorSystem;
import akka.stream.javadsl.BroadcastHub;
import akka.stream.javadsl.Keep;
import akka.stream.javadsl.MergeHub;
import akka.stream.javadsl.Sink;
import akka.stream.javadsl.Source;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;

class GreeterServiceImpl implements GreeterService {

  final ActorSystem<?> system;
  final Sink<HelloRequest, NotUsed> inboundHub;
  final Source<HelloReply, NotUsed> outboundHub;

  public GreeterServiceImpl(ActorSystem<?> system) {
    this.system = system;
    Pair<Sink<HelloRequest, NotUsed>, Source<HelloReply, NotUsed>> hub
      MergeHub.of(HelloRequest.class)
        .map(request ->
            HelloReply.newBuilder()
                .setMessage("Hello, " + request.getName())
                .build())
        .toMat(BroadcastHub.of(HelloReply.class), Keep.both())
        .run(system);

    inboundHub = hubInAndOut.first();
    outboundHub = hubInAndOut.second();
  }

  @Override
  public CompletionStage<HelloReply> sayHello(HelloRequest request) {
    return CompletableFuture.completedFuture(
```

```
        HelloReply.newBuilder()
            .setMessage("Hello, " + request.getName())
            .build()
    );
  }

  @Override
  public Source<HelloReply, NotUsed> sayHelloToAll(Source<HelloRequest
    in.runWith(inboundHub, system);
    return outboundHub;
  }
}
```

To connect all input and output streams of all connected clients dynamically we use a **MergeHub (https://doc.akka.io/docs/akka/current/stream/stream-dynamic.html#using-the-mergehub)** for the incoming messages and a **BroadcastHub (https://doc.akka.io/docs/akka/current/stream/stream-dynamic.html#using-the-broadcasthub)** for the outgoing messages.

The `MergeHub` and `BroadcastHub` are only needed because we want to connect different clients with each other. If each client was separate it might look like this to have the stream of incoming messages from one client transformed and emitted only to that client:

```
  public Source<HelloReply, NotUsed> sayHelloToAll(Source<HelloRequest
    return in.map(request ->
        HelloReply.newBuilder()
          .setMessage("Hello, " + request.getName())
          .build());
  }
```

Client

The client is emitting `HelloRequest` once per second and prints the streamed responses:

```
copyimport akka.Done;
import akka.NotUsed;
import akka.japi.Pair;
import akka.actor.typed.ActorSystem;
import akka.actor.typed.javadsl.Behaviors;
import akka.grpc.GrpcClientSettings;
import akka.stream.javadsl.Source;

import java.time.Duration;
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.CompletionStage;

import static akka.NotUsed.notUsed;
names.forEach(name -> {
  System.out.println("Performing streaming requests: " + name);

  Source<HelloRequest, NotUsed> requestStream =
      Source
          .tick(Duration.ofSeconds(1), Duration.ofSeconds(1), "tick")
          .zipWithIndex()
          .map(Pair::second)
          .map(i ->
              HelloRequest.newBuilder()
                  .setName(name + "-" + i)
                  .build())
          .mapMaterializedValue(ignored -> notUsed());

  Source<HelloReply, NotUsed> responseStream = client.sayHelloToAll(re

  CompletionStage<Done> done =
      responseStream.runForeach(reply ->
              System.out.println(name + " got streaming reply: " + rep
          system);

  done.whenComplete((reply, error) -> {
    if (error == null) {
```

```
      System.out.println("streamingBroadcast done");
    } else {
      System.out.println("Error streamingBroadcast: " + error);
    }
  });
```

Next: *Testing gRPC (testing.html)*