Proposal: Bidirectional/Random-Access Streams in Flight

Motivation

Currently, Flight is based around unidirectional streams of data. This serves many use cases well, in particular when the client is offloading *storage* of data, and the DoGet and DoPut methods reflect this. However, this leaves one use case unaccounted for, where the client wishes to offload *computation*, by uploading some Arrow data and receiving Arrow data in response. For instance, the service may distribute an expensive computation over multiple machines, or join the data with other datasets.

This use case could be handled by the current protocol with some workarounds:

- The client could issue a DoGet with a custom message indicating that the server should first retrieve data from some location in a storage system or another service. However, this assumes the data can easily be made available through a storage service - this may not be true since the service would have to understand the authentication model, be granted (temporary) permissions, and so on.
- The client could issue a DoGet and DoPut in parallel, linking them at the application level. However, such a design carries major pitfalls.
    - If a load balancer is present, for example, then the DoGet and DoPut may get routed to different instances. (gRPC is often handled by layer 7 load balancers that can do precisely this.)
    - This complicates server code, which must synchronize state across two separate logical RPC calls.
    - Such a design duplicates application metadata channels - the server can return data both as part of messages in the DoGet channel and independently in the DoPut channel.

Thus, we favor designing an explicit method for this purpose.

gRPC gives us the capability to have fully bidirectional streams like this. But this is a risk, as other, future transports for Flight may not support this. Additionally, such methods greatly complicate API design; with two independent channels, it's easy for application developers to accidentally block. Flight currently lacks asynchronous APIs, but implementing a bidirectional stream API would make the need for this more pressing.

Design

A new method would be added. (A better name for this would be much appreciated.)

```
  service FlightService {
      ...
+     rpc DoExchange(stream FlightData)
+         returns (stream FlightData)
```

```
 }
```

No other changes would be made.

The low-level APIs in each language would expose functionality similar to the following strawman:

```
// Not actually code or API for any particular binding
interface FlightStream {
    Status Write(Optional<RecordBatch> batch,
                 Optional<Buffer> metadata);
    Status Read(out RecordBatch batch, out Buffer metadata);
    void DoneWriting();
    void Cancel(Status reason);
}
Status FlightClient#doBidirectional(out FlightStream toServer,
                                    out FlightStream fromServer)
Status FlightService#doBidirectional(FlightStream toClient,
                                     FlightStream fromClient)
```

The actual APIs should be idiomatic for the language (for instance, they should be asynchronous in Java), but it should be clear that there may be either metadata or data (or both) in a given message.

Some high-level methods would be useful. The most common use case we envision is a put-then-get, which could be handled by simple wrapper methods, e.g. in Python:

```
FlightClient.do_request_response(
    Table table, Buffer initial_metadata
) -> Tuple[Table, Buffer]
```

API Example Strawman

```
import pyarrow as pa
import pyarrow.flight as flight

class ExpensiveCalculationService(flight.FlightServerBase):
     def do_bidirectional(self, context, reader, writer):
     while True:
          batches = []

          message = reader.read()
          if message == None:
```

```
                # Client called writer.done_writing
                break

            while message:
                batch, metadata = message
                if metadata.to_pybytes() == b'end_of_batch':
                    break
                batches.append(batch)
                message = reader.read()

            # Do expensive calculation with batches
            result = join_and_run_computation(batches)
            writer.write(result)
        writer.done_writing()
        reader.close()


def bidirectional_user(client: flight.FlightClient):
        descriptor = flight.FlightDescriptor.for_command(
        b'expensive_calculation')
        info = client.get_flight_info(descriptor)
        reader, writer = client.do_bidirectional(
        info.endpoints[0].ticket)

        # The data is batched (at a different level than Arrow
        # RecordBatches), so use metadata to indicate the
        # boundary between batches.
        writer.write(batch1)
        writer.write(batch2)
        writer.write(metadata=b'end_of_batch')

        batch1, metadata = reader.read()

        writer.write(batch3)
        writer.write(batch4)
        writer.write(metadata=b'end_of_batch')

        batch2, metadata = reader.read()

        writer.done_writing()

        result = pa.Table.from_batches(batch1, batch2)
```

```
reader.close()
```