Flight Protocol/API Proposals

Summary:

- Proposal 1:
  - In FlightData, add a bytes field for application-defined metadata.
  - In DoPut, change the return type to be streaming, and add a bytes field to PutResult for application-defined metadata.
- Proposal 2: In client/server APIs, add a call options parameter to control timeouts and provide access to the identity of the authenticated peer (if any).
- Proposal 3: Add an interface to define authentication protocols on the client and server, using the existing Handshake endpoint and adding a protocol-defined, per-call token.
- Proposal 4: Construct the client/server using builders to allow configuration of transport-specific options and open the door for alternative transports.

Application-level metadata in streaming calls
Use cases: retries/stream resumption

It would be useful to send per-message metadata both when the client is uploading and downloading. When downloading, the server could tag data with a progress marker or context data, so the client could resume the download if the connection is lost. (We have data that is not necessarily monotonically indexed, and which may need server-side state to reconstruct.) When uploading, the client could tag record batches with an identifier so that the server doesn't write duplicate data; additionally, the server could optionally send an acknowledgement at any point during upload, so that if the connection is lost, the client can use the acknowledgment to resume its upload from that point. The acknowledgment is not required to be sent.

Proposed design:
At the protocol layer, DoPut would now return a stream of PutResult. PutResult would contain a field of opaque bytes, as would FlightData. A Flight service could send back a result at any time, and does not have to ever send any message. We'd have to work out an appropriate API to send/retrieve this metadata for each language.

Per-call information: timeouts and authentication
Use cases: authentication, client/server robustness

The Flight protocol has an authentication endpoint, but it may not be flexible enough - it's designed around the assumption of a single token acquired at the start of a connection. And the service itself may want to know the identity of the client talking to it.

We'd also like to set a timeout for any call, and for streaming calls (DoGet on the client, DoPut on the server), also set an idle stream timeout. An "idle stream timeout" is for when there's

too long of a delay between messages on a streaming call. (You may have a long-lived operation for which you cannot set an overall timeout, but which you wish to cancel if you stop receiving data halfway, for instance.) These would help avoid blocking clients indefinitely. An immediate use case is health checks: we can implement a health check "action" in DoAction, but need a timeout so the check itself doesn't block indeterminately.

Proposed design:
Authentication would not affect the protocol layer, beyond the RPC call used for initial authentication, which already exists. (gRPC can pass headers independently of messages, for instance.) Flight implementations may choose to ignore timeouts, depending on the capabilities of the underlying RPC layer.

On the client, all calls would take an additional parameter (say, of type FlightCallOptions) containing timeout options. (We could add overloads to keep the existing signatures for convenience.)

On the server, all calls would receive an additional parameter containing authentication data and timeout options. This object would have methods to check if a call was cancelled or interrupted, so services can avoid doing unnecessary work. (This reflects gRPC, which requires explicit polling for this fact in some languages. Flight implementations may choose never to set this.)

For authentication, Clients and servers would be constructed with an authentication object, which exposes methods both called on the initial connection and each RPC. The initial connection method is expected to return whether authentication succeeded; it implements or calls the RPC method in the Flight protocol for this purpose. This method can send and receive multiple messages before reaching a result. The per-call method is expected to produce or consume an opaque binary string (an authentication token), and additionally return whether authentication succeeded. This supports both schemes that authenticate once at the start of a session, and ones that re-authenticate on every RPC.

Builders for server/client construction
Use cases: RPC-layer configuration

We'd like to be able to set options on the underlying transport, such as the timeout on how long a connection lives, or enabling/disabling instrumentation. While Flight can and should set reasonable defaults, applications may want to further tune these values. Additionally, we would like to support additional transports in the future, and need more flexible control over constructing transport-layer objects.

Examples of useful gRPC server/client options: max channel age/idle duration, keepalive time, using epoll-based event loop.

Proposed design:
Server/client constructors would be replaced by transport-specific builders, exposing a set of common options as well as transport-specific options. The builder would construct the high-level FlightClient or FlightServer object, which wraps the underlying implementation. A common base builder class would expose options available to all transports, and provide a generic interface for constructing clients/servers for a given Flight URI.

Other designs
We could expose the underlying gRPC service/channel objects via some sort of optional API (to avoid bloating dependents with more dependencies). This doesn't let us implement per-call timeouts, though. Also, it is not (reasonably) possible to expose a Python gRPC object from the Flight bindings. (Thus, just providing a "fromGrpcChannel" method wouldn't work, without also binding gRPC/C++ to Arrow/Python.)

As an aside - I think it would be interesting to sketch out a WebSockets implementation, perhaps in Python or Java, to validate that Flight's concepts are portable across RPC layers, and to work out any kinks where we've been too gRPC-specific. This might also enable interesting use cases for data visualization and data science, to be able to efficiently fetch Arrow data in the browser in a uniform way. Example: Wes pointed me to Falcon (https://github.com/uwdata/falcon).