7 Making the data available

This chapter covers:

- Common communication patterns
- When to use webhooks, HTTP Long Polling, server-sent events, and WebSockets
- Use case: building a Meetup RSVP streaming API

We have come a long way through our architecture and are now ready to consider how to deliver the data to a streaming consumer. When designing this tier, we are faced with a similar problem as with the other tiers—that there are myriad technologies we can choose from and many ways we can build it. The other tiers have dealt with ingesting data, moving it around, analyzing it, and getting it ready for use. Without a doubt they all have their challenges and are fun and exciting to build, but I find the data access tier the most rewarding and fun of all. It is where all our hard work pays off—by delivering data to a client application. The crux of this tier is that we enable our customers to build applications with real-time features using our API. Many benefits come from having access to this type of API. One that quickly comes to mind is customers may derive quick business value by being able to see and act on data in real time. Another is by enabling developers to build more compelling applications, you can improve the development experience and empower them.

Our goal in this chapter is to see how we can build a streaming data API, taking into account the various communication patterns, discuss how to handle failure when we are building it, and look at the numerous protocols available. Figure 7.1 shows where we are in relation to the overall architecture we're working our way through. We'll begin our discussion with communications patterns.

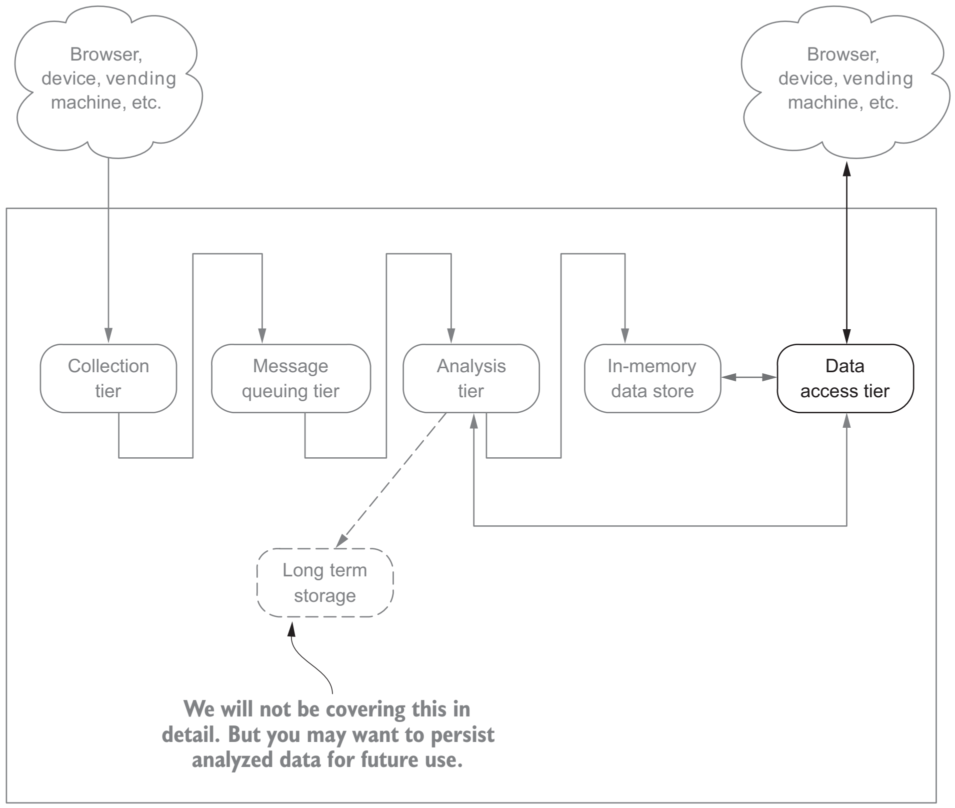
Browser, device, vending machine, etc.
Browser, device, vending machine, etc.
Collection tier
Message queuing tier
Analysis tier
In-memory data store
Data access tier
Long term storage
We will not be covering this in detail. But you may want to persist analyzed data for future use.

Figure 7.1 Overall streaming architecture blueprint with the chapter focus identified

7.1 Communications patterns

With a streaming system we're handling a continuous stream of data that is often being pushed at us, but when it comes to the communication patterns with clients, this often doesn't hold true. We need to think about supporting pushing data to a client and/or the client pulling data from our API. Beyond generic push or pull, there are four common patterns: Data Sync, RMI/RPC, Simple Messaging, and Publish-Subscribe. These are loosely based on the patterns you may see in the message queuing tier as well. Let's briefly walk through each.

7.1.1 Data Sync

It may seem like any communication between our API and the client involves syncing data, but with the Data Sync communication pattern, a database or data store is often synchronized between the API and the client. Figure 7.2 shows the flow of how this commonly works.

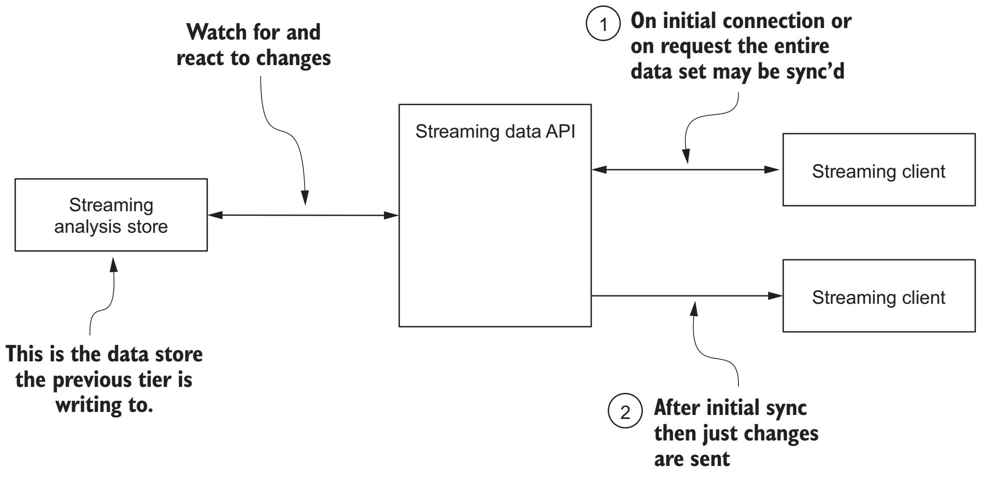
Watch for and react to changes
1 On initial connection or on request the entire data set may be sync'd
Streaming data API
Streaming analysis store
Streaming client
Streaming client
This is the data store the previous tier is writing to.
2 After initial sync then just changes are sent

Figure 7.2 Data Sync communication pattern showing initial and subsequent synchronization

The general flow of this pattern is that the streaming API watches (there may be a notification mechanism) for changes to the data store the analysis tier is writing to and then sends the updates to the streaming client. As shown in figure 7.2, there are usually two steps to this pattern. First, when the client initially connects—this may be when a mobile application is installed, for instance—the client will request the current data set. Subsequently, changes to the data set are either pushed to the client, or the client pulls the data. This pattern may seem simple enough, but there are benefits and drawbacks that you should consider before diving in.

Benefits

- Simple protocol.
- The client has a complete data set.
- The API is straightforward to develop because you are only supporting an initial sync and a delta after that based on either time or perhaps a version of the local data.
- The client always has a consistent view of the data that is the most current.

Drawbacks

- The data set may be large and require significant bandwidth to transfer.
- The data set may not fit on the device it is being transferred to.

- Need to resolve data version conflicts.
- Need to determine a merge policy so you know how to handle data changes made by the client with those made by the server.

Even with the drawbacks, for many applications this communication pattern may be beneficial and a great fit. For example, for a multiplayer mobile game application—where you want to always have the state of the game represented on every device without worrying about treating every move as an individual transaction—moving the current dataset may make sense.

7.1.2 Remote Method Invocation and Remote Procedure Call

With the Remote Method Invocation (RMI)/Remote Procedure Call (RPC) communication pattern, the API server invokes or calls a method on a connected client when new data arrives or when a condition is met that is of interest to the client. The general flow of this is depicted in figure 7.3.

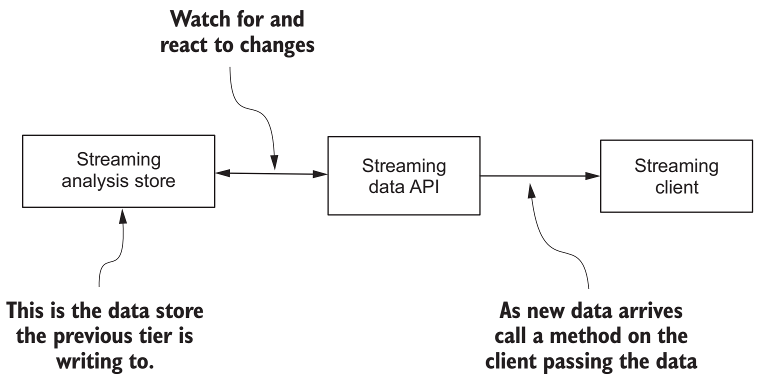
Watch for and react to changes
Streaming analysis store
Streaming data API
Streaming client
This is the data store the previous tier is writing to.
As new data arrives call a method on the client passing the data

Figure 7.3 Overall flow of RMI/RPC communication pattern

As you can see, the RMI/RPC communication pattern is fairly straightforward. In general, our API monitors the data store being written to by the analysis tier for changes and then sends them to the client via remote method calls. In this case, either data is being sent as part of the call, perhaps the latest value, or it may be a call to notify the client that a certain condition has been met.

BENEFITS

- Simple protocol.
- Client can perform other processing and then react when a handler is called.
- API is straightforward to develop—all you need is a way for a client to register an endpoint.

Drawbacks

- Detecting failures is hard—what if the client isn't available? How can the client know the server didn't receive new data?
- Frequency of updates may overwhelm a client.
- How does the API handle client failures?

7.1.3 Simple Messaging

With the Simple Messaging communication pattern the client initiates a request to the streaming API asking for the most recent data, and the API responds with the latest data. Without adding metadata to the request to indicate to the server "only return data if it is newer than X" or the streaming API letting the client know that the data won't change for X period of time, this pattern is inefficient because the client may be requesting data that hasn't changed since it last made the request. Continuously requesting the same data that hasn't changed is a waste of resources and an inefficient pattern. Later in this chapter, when we talk about the various protocols, you'll see how this simple pattern takes a form that makes it more efficient. Each form adds a little something to the request and/or the response so that the client doesn't make extra calls to the streaming API for data that hasn't changed. Figure 7.4 shows the general flow of how this pattern works.

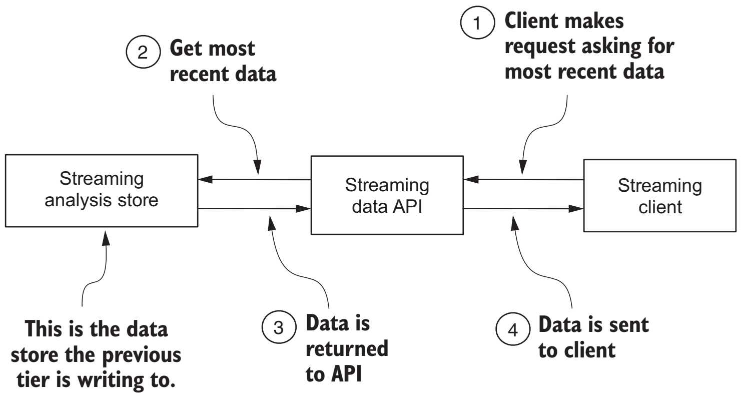
1 Client makes request asking for most recent data
2 Get most recent data
Streaming analysis store
Streaming data API
Streaming client
This is the data store the previous tier is writing to.
3 Data is returned to API
4 Data is sent to client

Figure 7.4 The Simple Message communication pattern with client asking for most recent data

As you can see in figure 7.4, the streaming client asks for the most recent data, the API queries the streaming analysis store, the data is returned to the Streaming Data API, then to the client. There can be subtle variations to this. For instance, the client may be able to make a request asking for changes from a certain time forward. In that case the API can check for changes and return the data changes since that point—or return no data at all if there were no changes since the time specified by the client.

BENEFITS

- Simple protocol and API call for the consumer to make.
- Only the most recent data is sent to the client.
- The client only has to keep track of a little metadata to continue getting the most recent data.
- The API doesn't have to keep track of any client state.

DRAWBACKS

- The protocol can be chatty because the client may continue making constant calls for the new data.
- There is no mechanism to alert the client to the existence of new data.
- The payload of new data may be large if the client was offline for an extended period of time or the velocity of the data is large.

7.1.4 Publish-Subscribe

With the Publish-Subscribe pattern the client subscribes to a particular channel, and the API then sends messages to all clients subscribed to that channel when the data changes. I'm using the term *channel* here to loosely represent the idea that the data can be grouped into categories or topics. The general flow of this pattern is shown in figure 7.5.

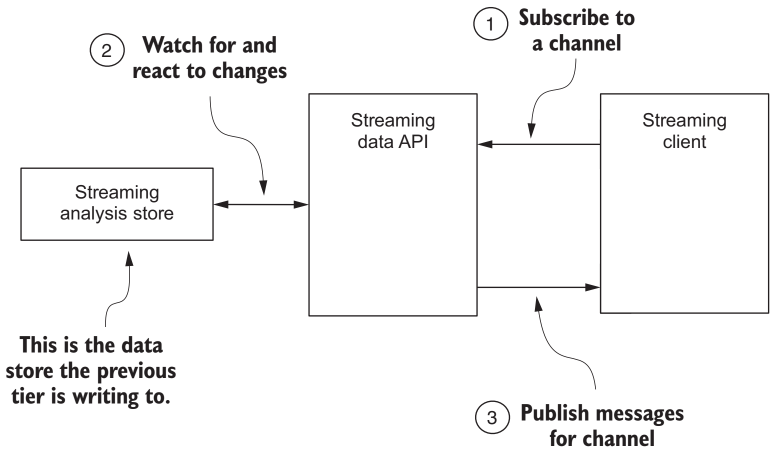
1 Subscribe to a channel
2 Watch for and react to changes
Streaming data API
Streaming client
Streaming analysis store
This is the data store the previous tier is writing to.
3 Publish messages for channel

Figure 7.5 The Publish-Subscribe communication pattern

If you're familiar with the Observer design pattern, you may recognize the data flow in figure 7.5. There is an object that keeps track of a list of dependents and state. When the state changes, the object notifies the list of dependents. With the Publish-Subscribe pattern, the streaming API maintains a list of streaming clients, and when the data changes in the streaming analysis store it notifies all the streaming clients.

This pattern has many advantages over some of the previous patterns. Perhaps the biggest advantage is that the streaming API can keep track of all the clients that are subscribed to a particular channel and then publish messages to all of them when there is new data. This reduces the burden on both the clients and the streaming API. The clients don't have to constantly call asking for new data or keep track of any metadata that would inform them of when new data may be available. The streaming API is also relieved of having to respond to client requests for data that hasn't changed. That may seem like it shouldn't be a burden for the streaming API, but think about the impact of millions of clients making requests for data that hasn't changed—that would be a waste of network resources and taxing to the streaming API to respond to.

This pattern is becoming more and more pervasive as we move toward a world of more reactive programming and streaming systems. If you use Twitter and follow a hashtag using a Twitter client, you've seen this pattern in action. Slack, the instant messaging application, also uses this pattern. If you haven't used Slack, the basic feature is this: Slack allows users to join a channel and then sends updates to each subscribed user as new messages are posted. You can probably think of many other examples of this pattern in use today.

Benefits

- The client can perform other processing and then react when data arrives.
- The client doesn't need to maintain any metadata about the current data.
- The API can optimize how it handles sending data to multiple clients.

Drawbacks

- There's a more complex protocol for the API.
- The API has to keep track of all metadata related to the clients, and must have the ability to distribute this across API servers in the event of a failure.

7.2 Protocols to use to send data to the client

Understanding the common patterns of communication is a great start. But before we can build our streaming API, we should look at the protocols that are commonly used and see which ones we may want to consider for our streaming API. We'll consider the following factors for each protocol:

- Message update frequency
- Direction of communication
- Message latency
- Efficiency
- Fault tolerance/reliability

7.2.1 Webhooks

Webhooks have been around since 2007. Though not an official W3C standard, they've been adopted by many as a way for a client to register a user-defined HTTP endpoint to be called when new data arrives or a condition is met. Conceptually, this is similar to the *callbacks* you may be familiar with in many programming languages. There are some stark differences, though. First, the callbacks in this case are executed via an HTTP POST. Secondly, often the client is implemented by a third-party developer. The general way in which this works is illustrated in figure 7.6.

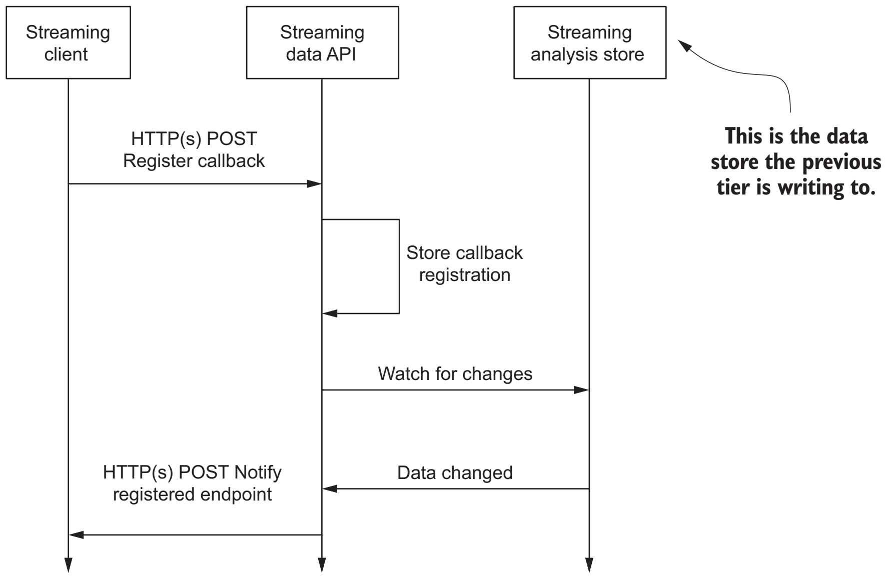
Streaming client
Streaming data API
Streaming analysis store
This is the data store the previous tier is writing to.
HTTP(s) POST Register callback
Store callback registration
Watch for changes
HTTP(s) POST Notify registered endpoint
Data changed

Figure 7.6 Webhook registration and callback

In the first step the client sends a request to register the callback via HTTP POST. Sometimes this step is done manually by a user filling the information in on a website. Subsequently the callback information is stored. After that, the streaming API calls all the registered callbacks when new data arrives or an event condition is met.

Now let's look at common factors:

- *Message update frequency*—Considering that every update is sent via an HTTP POST, it's fair to say that the frequency of sending updates would be low. There is nothing inherent in the protocol that would prevent a streaming API from calling it at a high rate, but considering the textual nature of the protocol and the inherent overhead, it's not ideal for high frequency.
- *Direction of communication*—The direction of communication is always from server to client. If any changes need to be made, it involves the client changing their registration. There is nothing standard about the registration process; it is 100% up to the provider of the streaming API.
- *Message latency*—The latency of messages is average. Many software stacks have highly tuned HTTP stacks, and although the protocol is text-based, we can take advantage of HTTP compression and chunking if the data updates are large.

- *Efficiency*—From the streaming API perspective this protocol is efficient. No state has to be maintained outside of the list of callback endpoints. When an update is to be sent (data or a condition), the server can make an asynchronous HTTP POST request to each of the registered callback endpoints.
- *Fault tolerance/reliability*—The protocol doesn't provide any guarantees—it's all up to us. We need to consider what to do if the HTTP POST fails. And because the communication is unidirectional, there's no way for the client to acknowledge that data was received. The HTTP protocol would allow us to make a separate GET request to see if a POST was successful, but that adds complexity because we now also need to handle failure of that request. If we choose to use this protocol to implement our streaming API, we have to address at least the following:
  - What do we do with the messages that were going to be sent to a client and the POST failed? Do we retry sending the message?
  - Is there a way for a client to get the messages that it would have missed if the HTTP POST failed?

Webhooks is a fairly simple protocol. Based on the discussion of the factors we're considering, it's not something that supports everything we would want in a streaming API. A common use case for webhooks is in systems with a low volume of messages that aren't impacted if a message is missed.

7.2.2 HTTP Long Polling

HTTP Long Polling involves the client making a connection to the server (in this case the streaming API server), the connection being held open, and the data being sent to the client as it is available. The general flow of control is shown in figure 7.7.

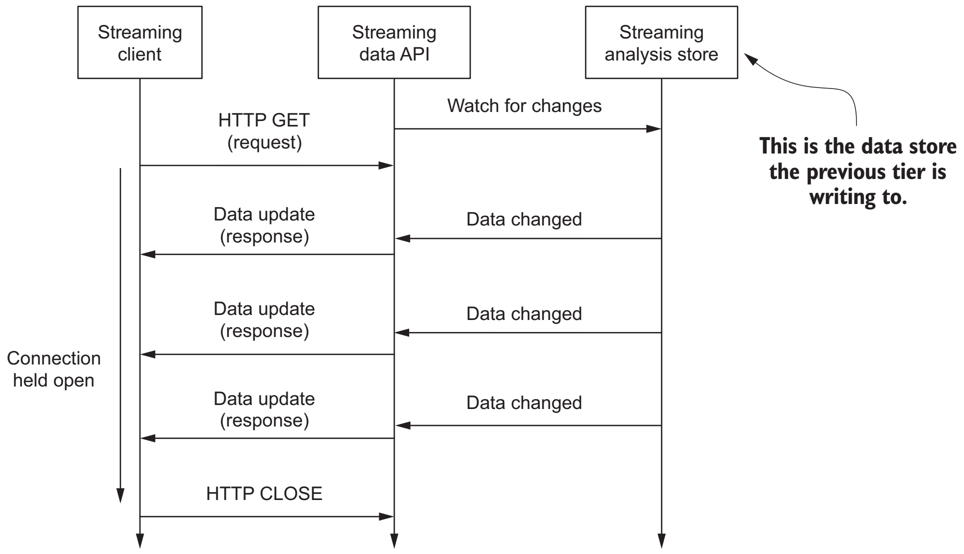

Figure 7.7 HTTP Long Polling flow of control

The client makes a request, and the server holds the connection open till there is an update and then sends it. The client in turn immediately opens a new connection, and the cycle repeats. This allows the client to control querying for data changes, but it comes at the cost of our streaming API being forced to keep open connections to all the clients.

Let's consider our common factors to see what else we need to keep in mind if we want to build our streaming API with this protocol:

- *Message update frequency*—Similar to the other HTTP-based protocols, there is nothing inherent in the communications protocol that would prevent using HTTP Long Polling for high update rates. Nevertheless, considering the textual nature of the protocol and the inherent overhead of a client getting a message, the connection being closed, and the client having to re-establish the connection, the overhead would become expensive if there was a high update frequency.
- *Direction of communication*—The client establishes the connection and can specify what it wants updates for, and the server responds.
- *Message latency*—As with webhooks, the latency of messages going over HTTP is average.
- *Efficiency*—Although the underlying HTTP protocol we are sending data over may be efficient, HTTP Long Polling isn't an efficient protocol. Our streaming API server will have clients keeping connections open to us while they wait for data and then immediately reopening them again as soon as they get a message. Our API server will be constrained by the number of open connections. We'll also need a way to ensure that if a client disconnects, the connection is closed. For example, when a mobile device rapidly switches between Wi-Fi and cellular networks or loses its connection, and its IP address changes, will a connection be automatically re-established?
- *Fault tolerance/reliability*—Similar to webhooks, this protocol doesn't provide any guarantees; it's all up to us. To implement our streaming API with this protocol we have to address at least the following:
  - We need to ensure that any new messages that arrive before the client reconnects are not lost.
  - To prevent a single streaming API host from getting overwhelmed, we need to understand how to load balance connections.
  - To prevent a client from losing or missing messages, we need to ensure we can handle the failure of our streaming API servers.

HTTP Long Polling is much closer to what we are looking for in a streaming API. It became popular with the rise of asynchronous programming in client-side development, rising in prominence with the use of Asynchronous JavaScript and XML (AJAX). The first web-based chat applications used HTTP Long Polling for real-time communication. It's still in use and is accessible to non-web-based clients as well because it's an HTTP request. It's supported by most if not all programming languages and is available on devices from servers to the small Raspberry Pi.

7.2.3 Server-sent events

Server-sent events (SSE) was developed and the subsequent W3C recommendation established in 2015 as an improvement over HTTP Long Polling. SSE helps solve at least two problems. First, it solves the inefficiency that exists when the client constantly closes and opens connections for every message received. Secondly, when using a resource-constrained device such as a mobile device it supports using a push proxy server, allowing the device to enter sleep mode while idle and be pushed messages from the proxy. This results in significant power savings for the device compared to keeping the connection open while idle. The two scenarios for SSEs are illustrated in figures 7.8 and 7.9.

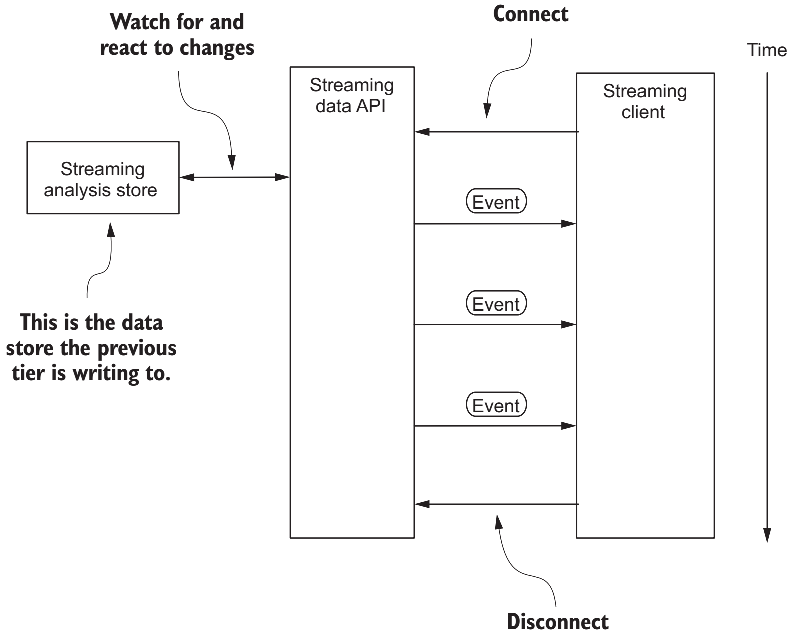

Figure 7.8 SSE data flow with a connected client

Figure 7.8 shows that the streaming client first establishes a connection to our streaming API server, and then as messages become available they are sent to the client. Unlike Long Polling, where the connection is closed and reopened for every message, in this case all the events are sent over the same connection. This results in more efficient network utilization and allows the client to do other processing while waiting for events to arrive. This still requires a client to maintain the connection, though. The second mode that SSE supports is *connectionless push*, illustrated in figure 7.9.

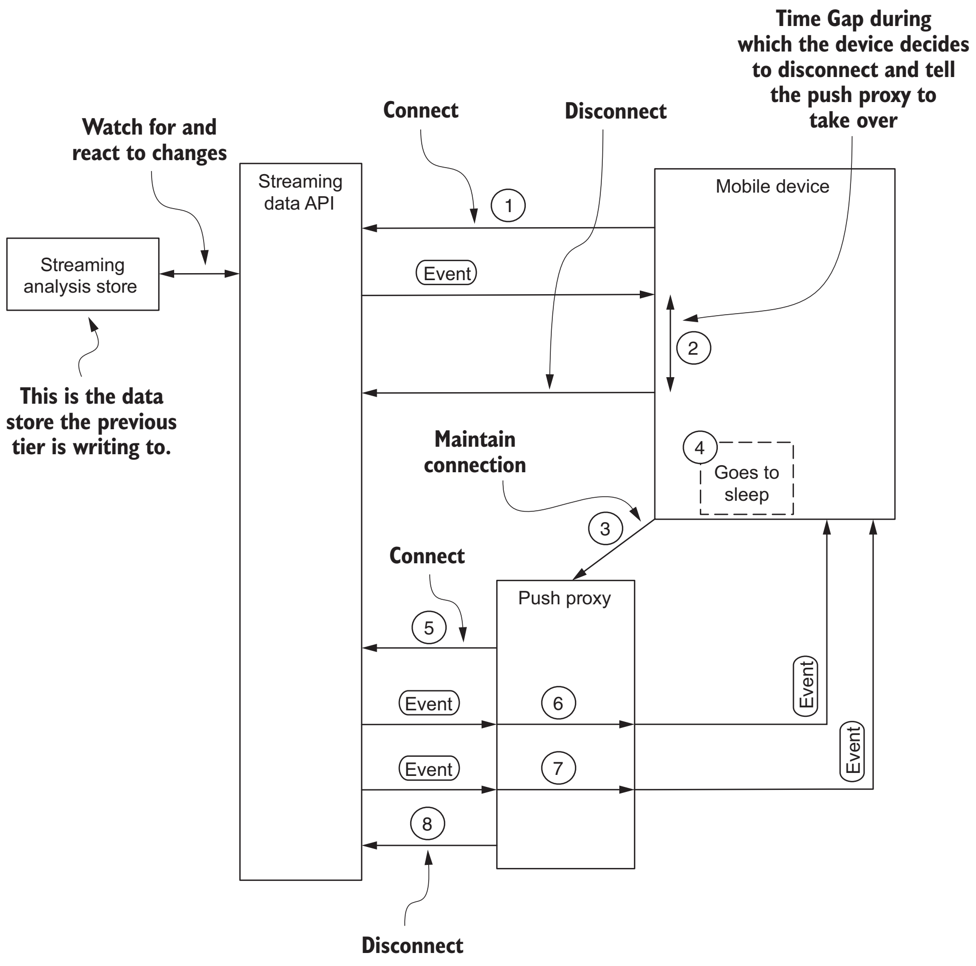
Time Gap during which the device decides to disconnect and tell the push proxy to take over
Connect
Disconnect
Watch for and react to changes
Streaming data API
Mobile device
1
Streaming analysis store
Event
2
This is the data store the previous tier is writing to.
Maintain connection
4
Goes to sleep
3
Connect
Push proxy
5
Event
6
Event
Event
7
Event
8
Disconnect

Figure 7.9 SSE showing connectionless data flow

The data flow shown in figure 7.9 is a little more complex than figure 7.8. Let's walk through it:

1. The client, in this case a browser on a mobile device, connects to the streaming API.
2. After receiving a single event, enough time goes by that the device decides (this is up to the implementer) that to save power it's going into sleep mode.
3. Before going into sleep mode the mobile device sends a message to the push proxy asking it to maintain the connection. It may send along the ID of the last event it received so the push proxy can pick up from there.
4. The mobile device sleeps to conserve power.
5. The push proxy establishes a connection to the streaming API.

6 When the push proxy receives an event, it sends it to the mobile device using a handset-specific push technology. The device wakes to process the message and then resumes sleeping.
7 This is the same flow as step 6.
8 At some point the push proxy disconnects from the streaming API, and the whole flow is finished.

For added complexity, using the push proxy, the mobile device can save power and reduce data usage using this workflow. The data use reduction and power savings are by-products of the fact that the push proxy is maintaining the connection with the API server, not the mobile device. The device doesn't pay the heavy cost of keeping a TCP connection alive, which can be expensive. When there is new data, the proxy uses a technology such as a push message to wake the device so it can receive and process the message and then go back to sleep. The result is that your users have a better experience when using your streaming API.

Considering how similar these two variants are, I'll treat them as one when looking at our common factors next, calling out any differences:

- *Message update frequency*—Although the messages with this protocol use HTTP as the transport protocol, the update frequency can be much higher than HTTP Long Polling. This increase in message update frequency is because that with this protocol there is a single persistent connection. You need to be cautious, though—you may run into problems with a client not being able to read data fast enough from the network at high message rates.
- *Direction of communication*—Outside of the initial connection, this protocol is clearly unidirectional, with the server pushing the events to the client.
- *Message latency*—As mentioned in previous HTTP protocol–based sections, the latency of messages going over HTTP is average.
- *Efficiency*—This protocol is as efficient as it can be given that it uses HTTP as the transport. There's a single connection per client over which all data is sent, eliminating the cost of constantly opening and closing connections. On the server side any gains in efficiency are in how fast data can be written to the socket, and care needs to be taken to ensure the client can read the data at the same rate off the wire. There is nothing inherent in this protocol that allows for the negotiation and managing of a maintainable message rate. Therefore, if you find that your clients can't keep up with the rate of messages, you may want to use a different protocol.
- *Fault tolerance/reliability*—Unfortunately, this protocol is similar to the previous HTTP–based protocols, where there are no guarantees. Because this protocol is unidirectional, from the server pushing to the client, we can't make this 100% fault tolerant and reliable from a message standpoint. But we can still address the following:
  - We can ensure that as new messages arrive we don't write them to the client if the network buffers are full. This would indicate the client can't read the

messages fast enough. With this protocol there's no way for the client to communicate that to the API; we have to rely on gathering this information from the network stack.
  - Because clients have persistent connections to the streaming API, it's important to ensure a single streaming API host doesn't get overwhelmed. Therefore, you need to know how to load balance connections.
  - To ensure that our streaming API doesn't miss sending messages to a client, we also need to ensure that when a streaming API server fails, another server can pick up sending messages from the last message sent forward. There are two sides to being able to do this: first, the streaming API must keep track of, in a distributed fashion, the last message ID sent to each client. When a server fails, the server that picks up the work can continue where the failed node stopped. Secondly, the client can help by sending the last event ID it received when it re-establishes a connection. An API server can then use this information to send messages from that point forward.

Overall this protocol yields better performance and allows us to provide a more resilient API. It is becoming the go-to alternative to HTTP Long Polling and is one you may want to consider as long as the fault tolerance and reliability concerns are acceptable to your use case.

Next we'll talk about WebSockets, which provides more flexibility than what we've seen so far.

7.2.4 WebSockets

WebSockets has been around since 2011. It's a full-duplex protocol that uses TCP for the communication transport. All major desktop and mobile browsers support it. Although it's commonly used between web clients and web servers, libraries are available for all major programming languages, allowing more interesting use cases to be solved. WebSockets is an interesting protocol in the sense that it uses HTTP for the initial handshake and protocol upgrade request and then switches to TCP. Figure 7.10 illustrates the WebSockets data flow.

Several things are going on with this protocol that are called out in figure 7.10:

1. This initial sequence of events—the handshake—happens between the client and the server over HTTP. It starts with the client initiating the handshake as an HTTP upgrade request. The server responds to the client, completing the handshake, and the protocol is upgraded from HTTP to TCP.
2. As events arrive, the server sends them to the client over TCP.
3. The "slow down" here is because the communication is bi-directional and the client may send a message to the server at any time. If a message is sent, what it contains and means is completely up to you. In the earlier example, we can imagine that the client can't read the data fast enough and is asking the server to slow down the frequency at which messages are sent.

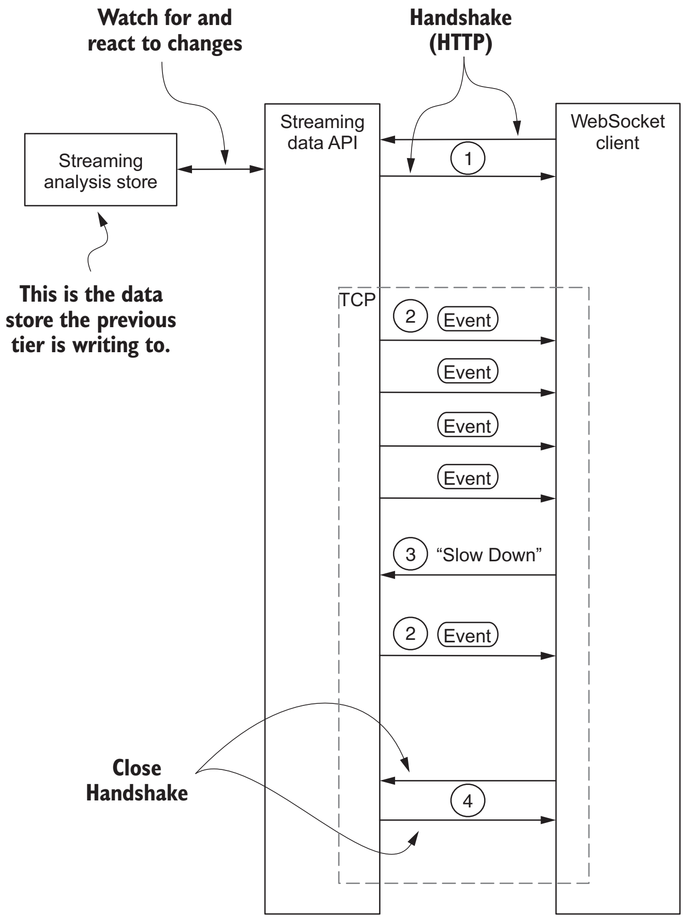

Figure 7.10 Example WebSocket data flow showing the client sending a message

4 At some point the client or server will want to end the connection. Although a connection may be abruptly closed, the correct way to take care of it is via *close handshake*. Figure 7.10 shows the client initiating the close, but the server can initiate it as well. Regardless of which side initiates it, the flow is the same: there is a close handshake that starts the close, followed by a close handshake response. At that point, the underlying TCP connection is closed.

See code listing 9.3.1

A lot is going on with this protocol, as you can tell. Let's look at the common factors to see how they can be addressed with this protocol:

- *Message update frequency*—Considering that all data for this protocol is sent over TCP, the message update frequency can be much higher than all the previous protocols we've looked at. Similar to SSEs, this increase is due to the persistent connection and in this case the protocol. There is the same risk with this protocol as there is with SSEs, that you need to be sure a client can keep up with the update frequency. The throttling of messages and slowing down isn't part of this protocol, but as shown in figure 7.10 this protocol lets you implement message-throttling semantics.

- *Direction of communication*—Unlike all previous protocols, this one is truly bi-directional from beginning to end, enabling you to create interesting subprotocols.
- *Message latency*—All communication for this protocol is over TCP, so latency is low compared to the previous HTTP-based protocols.
- *Efficiency*—Given that this protocol sends all data over TCP and there is a single persistent connection between a client and a server, this protocol is very efficient.
- *Fault tolerance/reliability*—Unfortunately, like the other protocols we've considered, this one doesn't offer any guarantees. But unlike the others this one is bi-directional, allowing us to build fault-tolerant and reliability semantics into it. In doing so, we need to consider how to handle the following:
  - If we are receiving events and trying to write them to a client faster than it can read them off the wire, we need to buffer them and not write them. Leveraging this protocol's bi-directional nature, we can make the client an active participant and have it send a message, perhaps something similar to "slow down," indicating that it needs the server to send messages at a slower rate.
  - Similar to SSEs, to ensure that our streaming API doesn't lose messages, we need to keep track of which message was sent to which client. Therefore, when a server fails and/or a client reconnects, the next message is the one expected and there's no gap in the messages sequence. Nothing inherent in the protocol provides this, but it's relatively easy to put in place. An advantage this protocol has over SSEs in this area is that with a little work you can have the clients send an acknowledgment to the server that not only did they receive the message, they have also processed it. This allows you to build a system that is highly fault tolerant and reliable.

After considering the common factors and the previous protocols we looked at, it may not come as a surprise that this protocol is more efficient, powerful, and flexible than the others. Here are some reasons for this:

- A single TCP connection is held open for as long as the client wants to consume data.
- The communication between client and server isn't HTTP, reducing the protocol overhead.
- The communication is bi-directional, enabling client and server to communicate over the single connection without having to open or close connections.
- We can implement fault tolerance and guarantee message delivery.

- Due to the lower level nature of the protocol, the streaming API and clients can maintain a higher level of throughput (measured in messages per second).
- This protocol allows us to develop a custom sub-protocol—the messages passed back and forth between the client and streaming API after the connection is established.

For streaming systems that use a client capable of handling HTTP, WebSockets is fast becoming one of the most-used protocols. It is the protocol to choose if you want to provide a fault-tolerant and reliable API. As mentioned, using this bi-directional protocol you can control the server and client sides of the communication. If you're building a system that uses resource-constrained clients that don't support HTTP, you may want to look at two other protocols: Data Distribution Service (DDS) and MQ Telemetry Transport (MQTT). Both were designed more for a Publish-Subscribe type model, but you may be able to use them for a stream of data.

We've covered a lot of ground talking about each of the protocols. Table 7.1 summarizes what we've talked about and can serve as a guide when you need to compare them.

Table 7.1 Summary comparisons of the different protocols

Protocol	Message frequency	Communication direction	Message latency	Efficiency	Fault tolerance/ Reliability
Webhooks	Low	Uni-directional (server to client)	Average	Low	None
HTTP Long Polling	Average	Bi-directional	Average	Average	None
Server-sent events	High	Uni-directional	Low	High	None by default. Has the ability to partially implement.
WebSockets	High	Bi-directional	Low	High	None by default. Has the ability to implement completely.

7.3 Filtering the stream

Before discussing aspects and nuances of filtering, let me define what I mean by *filtering*. Filtering in the context of the streaming API is the ability to restrict the events emitted and the properties of those events to only those of interest to a streaming client; this may be different than the filtering or reducing we learned about in our discussion of streaming algorithms. When considering filtering the stream, you need to understand the following:

- *Where the filtering is happening*—Analysis, streaming, or client tier (the client tier is our focus in the next chapter)
- *The type of filtering*—Static or dynamic

7.3.1 Where to filter

When it comes to deciding where to perform the filtering, you should consider several things. If any aggregations or other streaming algorithms are being applied to the stream, make sure all filtering happens in the analysis tier. Let's say we're interested in the hourly energy consumption along with the running total consumption for all homes in Chicago, IL, every 30 seconds. Given what we've learned, performing this type of computation in the analysis tier would be easy, and that would be the most appropriate place to perform that filtering. In some cases, if the volume of events/sec is low, an analysis tier may be overkill, in which case you may be able to do the filtering in the streaming API tier. Keep in mind that as the volume grows and/or the computations become more complex, you will want to look at adding an analysis tier. A use case where performing the filtering in the streaming API tier may make sense is if the stream you're producing is raw or slightly augmented, meaning the messages being emitted look the same or are close to the same as those ingested into the pipeline.

Imagine we have a stream of GPS location and vehicle metadata for all cars on highways in the United States. A streaming client may want to filter this data down to the events occurring in a certain geo-location, or by other metadata—for example, car make and model. This type of filtering would work fine in the streaming API tier. As the data is flowing through, we would filter out the events that did not match the criteria.

That brings up an important point: a user (streaming client) will be interested in both filtering whole events and in filtering out different properties from the events. If you're thinking this sounds a lot like SQL, you're correct, and that's a good way to think about it. A SQL `select` clause provides a way to retrieve only the data you want, using a `where` clause and filtering the data down to only the columns of data you are interested in. This is exactly what your users will want; conceptually the table in the relational world is now a stream, and the columns are properties on events.

The next section considers the two types of filtering.

7.3.2 Static vs. dynamic filtering

Filtering types fall into two buckets: static and dynamic. *Static* filtering is where the decisions about what the stream will contain are made ahead of time. You may think of this as a canned or out-of-the-box stream. It's the one that you as the streaming platform developer or architect decide on, and the client can't change the filtering being applied. From the relational standpoint this is similar to a view—the designer/developer of it decides on the data, and the user can't change it; it's the same for everyone.

Dynamic filtering is where the filtering is decided at run-time, and the streaming client can drive it. From a relational standpoint, the dynamic filtering is akin to running an arbitrary query against a table. In the streaming world, the "table" is the union of all the event schemas in the stream, giving the streaming client a lot of power. Given these different approaches, you'll also want to determine how to implement it with the protocol you chose or are considering for your streaming API.

Table 7.2 shows potential ways to think about integrating the different types of filtering for each of the protocols we've discussed.

Table 7.2 Considerations for filtering with the different protocols

Protocol	Dynamic filtering	Static filtering
Webhooks	When the endpoint is registered, you'd need a way to capture a query that can be used.	When the endpoint is registered, you'd need a way to capture a query that can be used.
HTTP Long Polling	You'd need to provide a means for the query to be expressed in a URL; the easiest thing is perhaps the URL itself representing the filter, such as '/top/50/products/viewed.'	You'd need to provide a means for the query to be expressed in a URL, perhaps using query parameters.
Server-sent events	You'd need to provide a way for the user to encode the filtering when they specify the URL for the EventSource constructor, perhaps in a similar fashion to HTTP Long Polling.	Similar to HTTP Long Polling, you'd need to provide a means for a user to specify the query as query parameters for the constructor of the EventSource.
WebSockets	This protocol provides the most flexibility. One way to handle this is after connecting, the client can send a message indicating the filtering; in the case of static, it can be a name.	One way to handle this is after connecting, the client can send a message indicating the query to be used for filtering.

As you can see, each protocol provides a way to communicate the desired filtering—the key difference is the level of flexibility you have in supporting it. When considering the design of your streaming API, remember that many developers and business people are familiar with SQL. Therefore, as you build your APIs and think through your filtering design, give some thought to how you can leverage their SQL knowledge and adopt an SQL-like syntax to your filtering. There are ways to add SQL-like query capabilities to your API. If you're building your pipeline using a JVM-based language, one option to explore is Apache Calcite (https://calcite.apache.org). Even if you're using a different language, this project may provide some ideas while you also evaluate SQL-parsing options for your chosen platform.

7.4 Use case: building a Meetup RSVP streaming API

Let's come back up for air and apply what we've learned to a use case. Let's pretend that we want to build a streaming system that uses the Meetup.com RSVP data as the data source. You can see this data source in action at http://stream.meetup.com/2/rsvps. At this URL you see a constant stream of JSON data; each entry is the result of someone entering an RSVP for a Meetup.

From this data source, imagine that we want to allow users to connect to our streaming API and filter the data by topic name. Because we're talking about the streaming API in this chapter, let's focus on making the data available to consumers.

In the second part of this book, when we build a real solution, we'll have to address how to consume the data from an analysis store. For now, we'll only worry about the communication and protocol choices we need to make. Figure 7.11 gives a depiction of what we're trying to accomplish.

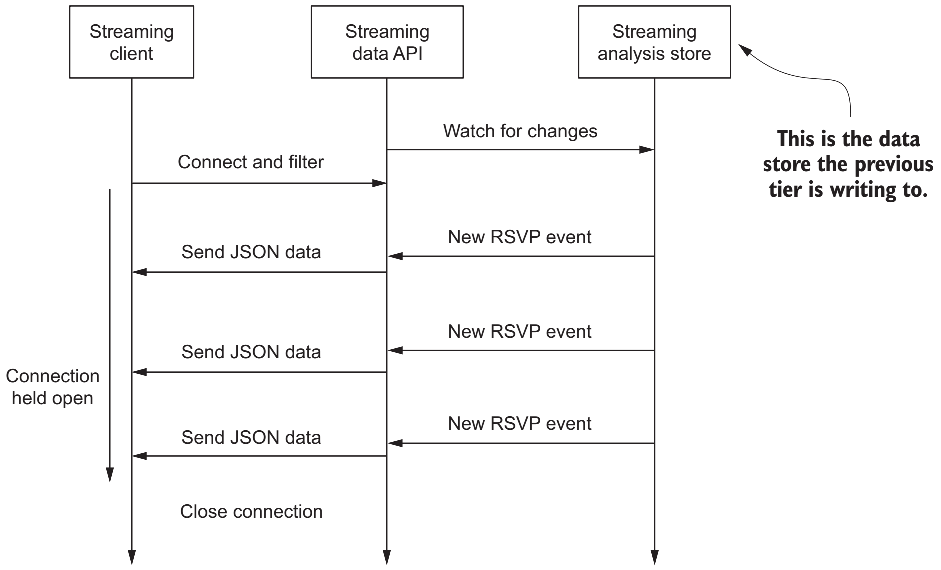

Figure 7.11 Meetup RSVP streaming API data flow with filtering when connecting

Figure 7.11 starts with a client connecting to our streaming API and passing in a filter. Our streaming API watches the analysis store for new data and then sends all new "RSVP Events" to the client. The two questions we need to answer are which communication pattern to use and which protocol to use? Tables 7.3 and 7.4 outline the communication and protocol choices, along with their applicability to this use case.

Table 7.3 Which communication pattern to use?

Communication pattern	Good fit	Comments
Data Sync	No	In this use case we're not transferring a complete data set, and the fact the data is continuously flowing would make the Data Sync protocol a bad choice.
RMI/RPC	No	Using RMI/RPC would not be a good fit here because we're looking at the data flowing toward the consumer and not having to call a method on the consumer each time a new event arrives.

Table 7.3 Which communication pattern to use? *(continued)*

Communication pattern	Good fit	Comments
Simple Messaging	Maybe	This pattern would not be a great fit because we're trying to do more than return the most recent data. In this case we're also applying filters to it. Although it may be something that can be made to work, there will be a lot of potential overhead on the API server to implement it.
Publish-Subscribe	Yes	This would be a good choice for this use case. We could gain some efficiency with the different protocols that can be used with this pattern, and the model fits how the data is flowing.

It looks like we have two communication patterns that would be candidates for this use case. Now let's look at the protocols.

Table 7.4 Which protocol to use?

Protocol	Good fit	Comments
Webhooks	No	There would be a lot of overhead because the streaming API would need to make an HTTP POST request to the registered callback for every new event. Not a good fit.
HTTP Long Polling	Yes	This protocol would work fine in this use case because our filtering is being sent during the client connection. To make that work with this protocol, we would want to allow the client to use a query parameter on the URL to indicate how they want to filter the results.
Server-sent events	Yes	This protocol would work well, given the same caveat as with HTTP Long Polling, that we're allowing the user to pass in the filter when they connect via a query parameter.
WebSockets	Yes	This protocol is a good choice as well. As with the other protocols that are good for this use case, we would need to have a way for the client to pass in the filter criteria. Where this protocol differs is that it can be a query parameter or a message that is sent when the client initiates the connection.

Looking at the communication and protocol options for this use case, we have a number of options that would allow us to build a streaming API that meets our needs. You should take several things into consideration when deciding which communication pattern and protocols to support. First, consider the clients you want to support and how you want the filtering to be performed. To allow the largest number of clients, providing Long Polling, server-sent events, and WebSockets would be the way to go. But if what is more important to you is that you allow filtering after the connection is established, then you need to use a more robust protocol such as WebSockets.

7.5 Summary

In this chapter we looked at the communications patterns and protocols used for sending data to a streaming client. We also talked about filtering a stream of data. We didn't dive too deeply into the different ways of implementing fault tolerance because they are all similar to what you've seen in the other chapters leading up to this.

Key take-aways include:

- It's important to closely look at your requirements and match up the different communication patterns and protocols. There is no one-size-fits-all prescription.
- When choosing the communication pattern and protocol, pay particular attention to your needs for fault tolerance and reliability, because they will drive your choices.
- Don't overlook supporting static and/or dynamic filtering. You may not think about them at first, but filtering is a feature your clients will want soon after your API is available.

Some of the protocols may not cover exactly what you need for your business problem, but I hope at least that after this chapter, you feel comfortable looking at other protocols and have a better understanding of how to approach looking at a new protocol.