1 Introducing streaming data

This chapter covers

- Differences between real-time and streaming data systems
- Why streaming data is important
- The architectural blueprint
- Security for streaming data systems

Data is flowing everywhere around us, through phones, credit cards, sensor-equipped buildings, vending machines, thermostats, trains, buses, planes, posts to social media, digital pictures and video—and the list goes on. In a May 2013 report, Scandinavian research center Sintef estimated that approximately 90% of the data that existed in the world at the time of the report had been created in the preceding two years. In April 2014, EMC, in partnership with IDC, released the seventh annual Digital Universe study (www.emc.com/about/news/press/2014/20140409-01.htm), which asserted that the digital universe is doubling in size every two years and would multiply 10-fold between 2013 and 2020, growing from 4.4 trillion gigabytes to 44 trillion gigabytes. I don't know about you, but I find those numbers hard to comprehend and relate to. A great way of putting that in perspective also comes from

that report: today, if a byte of data were a gallon of water, in only 10 seconds there would be enough data to fill an average home. In 2020, it will only take 2 seconds.

Although the notion of Big Data has existed for a long time, we now have technology that can store all the data we collect and analyze it. This does not eliminate the need for using the data in the correct context, but it is now much easier to ask interesting questions of it, make better and faster business decisions, and provide services that allow consumers and businesses to leverage what is happening around them.

We live in a world that is operating more and more in the *now*—from social media, to retail stores tracking users as they walk through the aisles, to sensors reacting to changes in their environment. There is no shortage of examples of data being used today as it happens. What is missing, though, is a shared way to both talk about and design the systems that will enable not merely these current services but also the systems of the future.

This book lays down a common architectural blueprint for how to talk about and design the systems that will handle all the amazing questions yet to be asked of the data flowing all around us. Even if you've never built, designed, or even worked on a real-time or Big Data system, this book will serve as a great guide. In fact, this book focuses on the big ideas of streaming and real-time data. As such, no experience with streaming or real-time data systems is required, making this perfect for the developer or architect who wants to learn about these systems. It's also written to be accessible to technical managers and business decision makers.

To set the stage, this chapter introduces the concepts of streaming data systems, previews the architectural blueprint, and gets you set to explore in-depth each of the tiers as we progress. Before I go over the architectural blueprint used throughout the book, it's important that you gain an understanding of real-time and streaming systems that we can build upon.

1.1 What is a real-time system?

Real-time systems and *real-time computing* have been around for decades, but with the advent of the internet they have become very popular. Unfortunately, with this popularity has come ambiguity and debate. What constitutes a real-time system?

Real-time systems are classified as *hard*, *soft*, and *near*. The definitions I use in this book for *hard* and *soft real-time* are based on Hermann Kopetz's book *Real-Time Systems* (Springer, 2011). For *near real-time* I use the definition found in the Portland Pattern Repository's Wiki (http://c2.com/cgi/wiki?NearRealTime). For an example of the ambiguity that exists, you don't need to look much further than Dictionary.com's definition: "Denoting or relating to a data-processing system that is slightly slower than real-time." To help clear up the ambiguity, table 1.1 breaks out the common classifications of real-time systems along with the prominent characteristics by which they differ.

You can identify hard real-time systems fairly easily. They are almost always found in embedded systems and have very strict time requirements that, if missed, may result

Table 1.1 Classification of real-time systems

Classification	Examples	Latency measured in	Tolerance for delay
Hard	Pacemaker, anti-lock brakes	Microseconds–milliseconds	None—total system failure, potential loss of life
Soft	Airline reservation system, online stock quotes, VoIP (Skype)	Milliseconds–seconds	Low—no system failure, no life at risk
Near	Skype video, home automation	Seconds–minutes	High—no system failure, no life at risk

in total system failure. The design and implementation of hard real-time systems are well studied in the literature, but are outside the scope of this book. (If you are interested, check out the previously mentioned book by Hermann Kopetz.)

Determining whether a system is soft or near real-time is an interesting exercise, because the overlap in their definitions often results in confusion. Here are three examples:

- Someone you are following on Twitter posts a tweet, and moments later you see the tweet in your Twitter client.
- You are tracking flights around New York using the real-time Live Flight Tracking service from FlightAware (http://flightaware.com/live/airport/KJFK).
- You are using the NASDAQ Real Time Quotes application (www.nasdaq.com/quotes/real-time.aspx) to track your favorite stocks.

Although these systems are all quite different, figure 1.1 shows what they have in common.

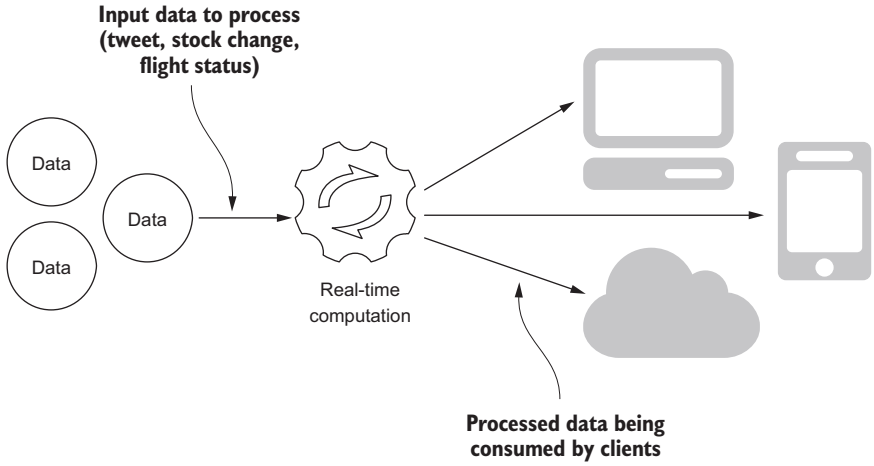

Figure 1.1 A generic real-time system with consumers

In each of the examples, is it reasonable to conclude that the time delay may only last for seconds, no life is at risk, and an occasional delay for minutes would not cause total system failure? If someone posts a tweet, and you see it almost immediately, is that soft or near real-time? What about watching live flight status or real-time stock quotes? Some of these can go either way: what if there were a delay in the data due to slow Wi-Fi at the coffee shop or on the plane? As you consider these examples, I think you will agree that the line differentiating soft and near real-time becomes blurry, at times disappears, is very subjective, and may often depend on the consumer of the data.

Now let's change our examples by taking the consumer out of the picture and focusing on the services at hand:

- A tweet is posted on Twitter.
- The Live Flight Tracking service from FlightAware is tracking flights.
- The NASDAQ Real Time Quotes application is tracking stock quotes.

Granted, we don't know how these systems work internally, but the essence of what we are asking is common to all of them. It can be stated as follows:

> Is the process of receiving data all the way to the point where it is ready for consumption a soft or near real-time process?

Graphically, this looks like figure 1.2.

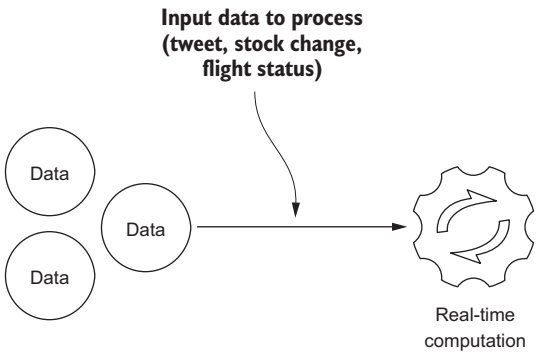

Figure 1.2 A generic real-time system with no consumers

Does focusing on the data processing and taking the consumers of the data out of the picture change your answer? For example, how would you classify the following?

- A tweet posted to Twitter
- A tweet posted by someone whom you follow and your seeing it in your Twitter client

If you classified them differently, why? Was it due to the lag or perceived lag in seeing the tweet in your Twitter client? After a while, the line between whether a system is soft

or near real-time becomes quite blurry. Often people settle on calling them real-time. In this book, I aim to provide a better way to identify these systems.

1.2 Differences between real-time and streaming systems

It should be apparent by now that a system may be labeled soft or near real-time based on the perceived delay experienced by consumers. We have seen, with simple examples, how the distinction between the types of real-time system can be hard to discern. This can become a larger problem in systems that involve more people in the conversation. Again, our goal here is to settle on a common language we can use to describe these systems. When you look at the big picture, we are trying to use one term to define two parts of a larger system. As illustrated in figure 1.3, the end result is that it breaks down, making it very difficult to communicate with others with these systems because we don't have a clear definition.

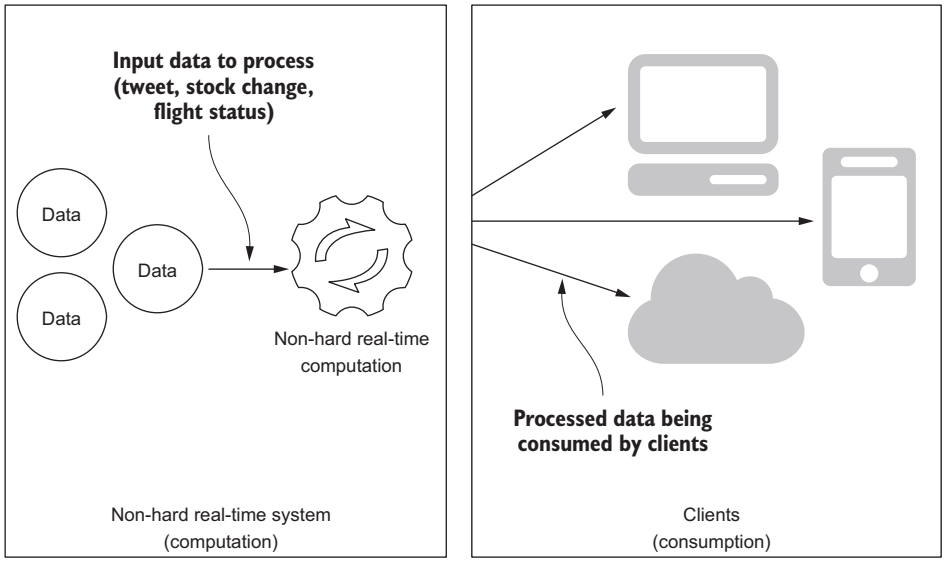

Figure 1.3 Real-time computation and consumption split apart

On the left-hand side of figure 1.3 we have the non-hard real-time service, or the *computation* part of the system, and on the right-hand side we have the clients, called the *consumption* side of the system.

DEFINITION: STREAMING DATA SYSTEM In many scenarios, the computation part of the system is operating in a non-hard real-time fashion, but the clients may not be consuming the data in real time due to network delays, application design, or a client application that isn't even running. Put another way, what we have is a non-hard real-time service with clients that consume data when they need it. This is called a *streaming data system*—a non-hard real-time

system that makes its data available at the moment a client application needs it. It's neither soft nor near—it is streaming.

Figure 1.4 shows the result of applying this definition to our example architecture from figure 1.3.

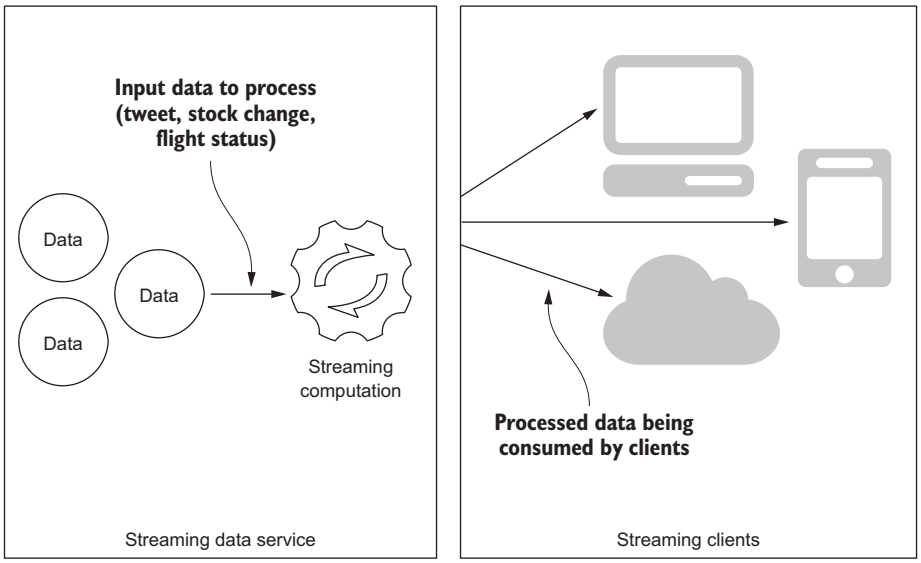

Figure 1.4 A first view of a streaming data system

The concept of streaming data eliminates the confusion of soft versus near and real-time versus not real-time, allowing us to concentrate on designing systems that deliver the information a client requests at the moment it is needed. Let's use our examples from before, but this time think about them from the standpoint of streaming. See if you can split each one up and recognize the streaming data service and streaming client.

- Someone you are following on Twitter posts a tweet, and moments later you see the tweet in your Twitter client.
- You are tracking flights around New York using the real-time Live Flight Tracking service from FlightAware.
- You are using the NASDAQ Real Time Quotes application to track your favorite stocks.

How did you do? Here is how I thought about them:

- *Twitter*—A streaming system that processes tweets and allows clients to request the latest tweets at the moment they are needed; some may be seconds old, and others may be hours old.
- *FlightAware*—A streaming system that processes the most recent flight status data and allows a client to request the latest data for particular airports or flights.

- *NASDAQ Real Time Quotes*—A streaming system that processes the price quotes of all stocks and allows clients to request the latest quote for particular stocks.

Did you notice that doing this exercise allowed you to stop worrying about soft or near real-time? You got to think and focus on what and how a service makes its data available to clients at the moment they need it. Thinking about it this way, you can say that the system is an *in-the-moment* system—any system that delivers the data at the point in time when it is needed. Granted, we don't know how these systems work behind the scenes, but that's fine. Together we are going to learn to assemble systems that use open source technologies to consume, process, and present data streams.

1.3 The architectural blueprint

With an understanding of real-time and streaming systems in general under our belt, we can now turn our attention to the architectural blueprint we will use throughout this book. Throughout our journey we are going to follow an architectural blueprint that will enable us to talk about all streaming systems in a generic way—our pattern language. Figure 1.5 depicts the architecture we will follow. Take time to become familiar with it.

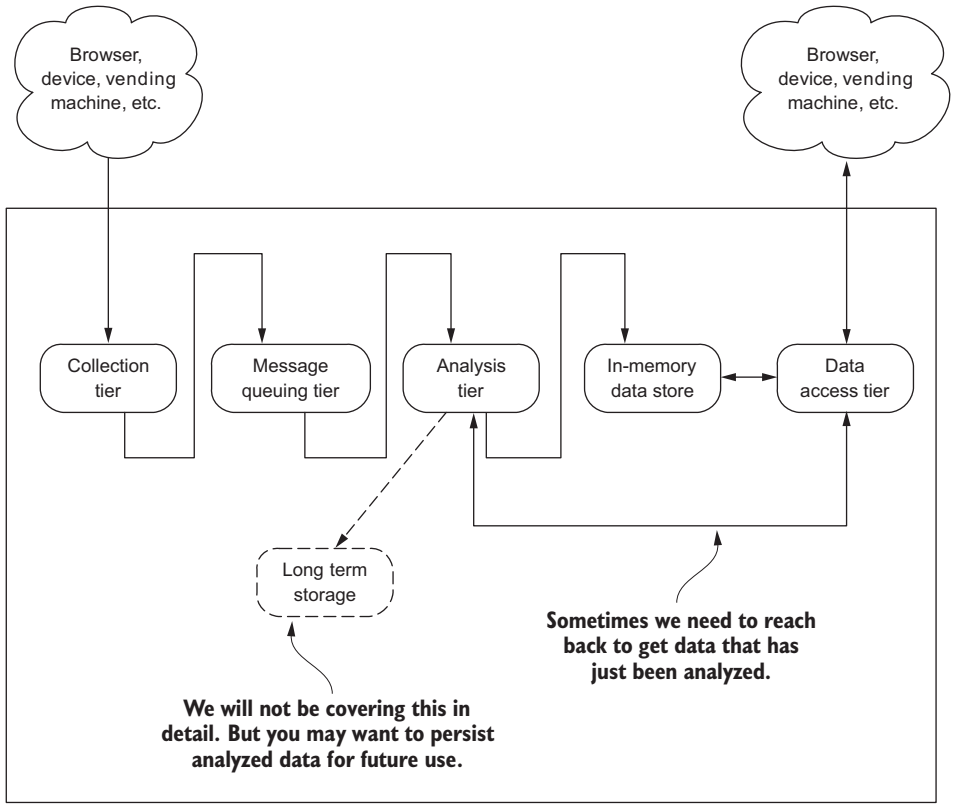

Figure 1.5 The streaming data architectural blueprint

As we progress, we will zoom in and focus on each of the tiers shown in figure 1.5 while also keeping the big picture in mind. Although our architecture calls out the different tiers, remember these tiers are not hard and rigid, as you may have seen in other architectures. We will call them tiers, but we will use them more like LEGO pieces, allowing us to design the correct solution for the problem at hand. Our tiers don't prescribe a deployment scenario. In fact, they are in many cases distributed across different physical locations.

Let's take our examples from before and walk through how Twitter's service maps to our architecture:

- *Collection tier*—When a user posts a tweet, it is collected by the Twitter services.
- *Message queuing tier* —Undoubtedly, Twitter runs data centers in locations across the globe, and conceivably the collection of a tweet doesn't happen in the same location as the analysis of the tweet.
- *Analysis tier*—Although I'm sure a lot of processing is done to those 140 characters, suffice it to say, at a minimum for our examples, Twitter needs to identify the followers of a tweet.
- *Long-term storage tier*—Even though we're not going to discuss this optional tier in depth in this book, you can probably guess that tweets going back in time imply that they're stored in a persistent data store.
- *In-memory data store tier*—The tweets that are mere seconds old are most likely held in an in-memory data store.
- *Data access*—All Twitter clients need to be connected to Twitter to access the service.

Walk yourself through the exercise of decomposing the other two examples and see how they fit our streaming architecture:

- *FlightAware*—A streaming system that processes the most recent flight status data and allows a client to request the latest data for particular airports or flights.
- *NASDAQ Real Time Quotes*—A streaming system that processes the price quotes of all stocks and allows clients to request the latest quote for particular stocks.

How did you do? Don't worry if this seemed foreign or hard to break down. You will see plenty more examples in the coming chapters. As we work through them together, we will delve deeper into each tier and discover ways that these LEGO pieces can be assembled to solve different business problems.

1.4 Security for streaming systems

As you reflect on our architectural blueprint, you may notice that it doesn't explicitly call out security. Security is important in many cases, but it can be overlaid on this architecture naturally. Figure 1.6 shows how security can be applied to this architecture.

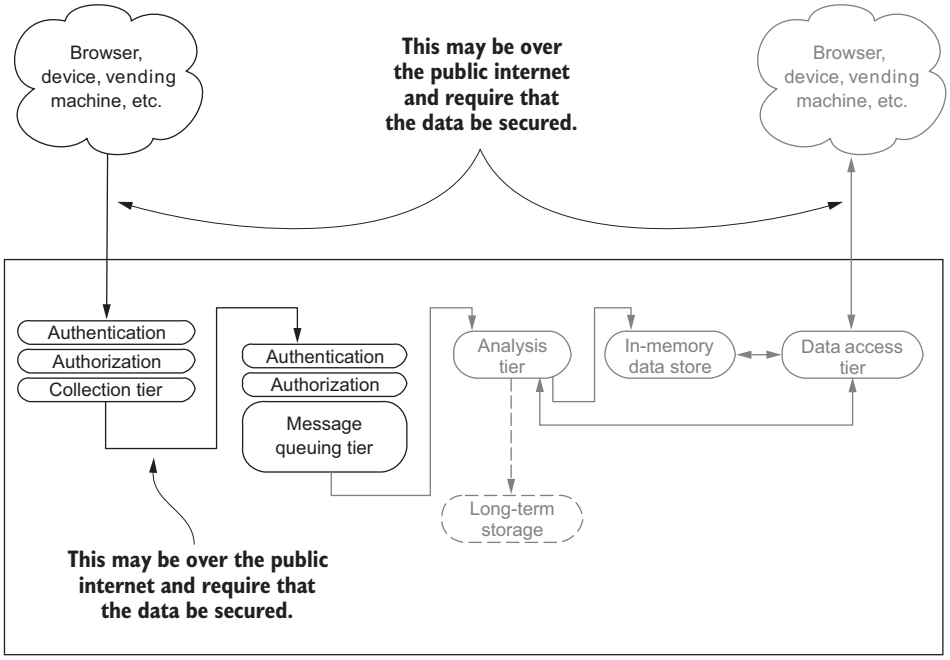

Figure 1.6 The architectural blueprint with security identified

We won't be spending time discussing security in depth, but along the way I will call it out so you can see how it fits and think about what it may mean for the problems you're solving. If you're interested in taking a deeper look at security and distributed systems, see Ross Anderson's *Security Engineering: A Guide to Building Dependable Distributed Systems* (Wiley, 2008). This book also is available for free at www.cl.cam.ac.uk/~rja14/book.html.

1.5 How do we scale?

From a high level, there are two common ways of scaling a service: vertically and horizontally.

Vertical scaling lets you increase the capacity of your existing hardware (physical or virtual) or software by adding resources. A restaurant is a good example of the limitations of vertical scaling. When you enter a restaurant, you may see a sign that tells you the maximum occupancy. As more patrons come in, more tables may be set up and more chairs added to accommodate the crowd—this is scaling vertically. But when the maximum capacity is reached, you can't seat any more customers. In the end, the capacity is limited by the size of the restaurant. In the computing world, adding more memory, CPUs, or hard drives to your server are examples of vertical scaling. But as with the restaurant, you're limited by the maximum capacity of the system, physical or virtual.

Horizontal scaling approaches the problem from a different angle. Instead of continuing to add resources to a server, you add servers. A highway is a good example of horizontal scaling. Imagine a two-lane highway that was originally constructed to handle 2,000 vehicles an hour. Over time more homes and commercial buildings are built along the highway, resulting in a load of 8,000 vehicles per hour. As you might imagine (and perhaps have experienced), the results are terrible traffic jams during rush hour and overall unpleasant commutes. To alleviate these issues, more lanes are added to the highway—now it is horizontally scaled and can handle the traffic. But it would be even more efficient if it could expand (add lanes) and contract (remove lanes) based on traffic demands. At an airport security checkpoint, when there are few travelers TSA closes down screening lines, and when the volume increases they open lines up. If you're hosting your service with one of the major cloud providers (Google, AWS, Microsoft Azure), you may be able to take advantage of this elasticity—a feature they often call *auto-scaling*. The basic idea is that as demand for your service increases, servers are automatically added, and as demand decreases, servers are removed.

In modern-day system design, our goal is to have horizontal scaling—but that doesn't mean that we won't use vertical scaling too. Vertical scaling is often employed to determine an ideal resource configuration for a service, and then the service is scaled out. But in this book, when the topic of scaling comes up, the focus will be on horizontal, not vertical scaling.

1.6 Summary

Now that you have an idea of the architectural blueprint, let's see where we have been:

- We defined a real-time system.
- We explored the differences between real-time and streaming (in-the-moment) systems.
- We developed an understanding of why streaming is important.
- We laid out an architectural blueprint.
- We discussed where security for streaming systems fits in.

Don't worry if some of this is slightly fuzzy at this point, or if teasing apart the different business problems and applying the blueprint seems overwhelming. I will walk through this slowly over many different examples in the coming chapters. By the end, these concepts will seem much more natural.

We are now ready to dive into each of the tiers to find out what they're composed of and how to apply them in the building of a streaming data system. Which tier should we tackle first? Take a look at a slightly modified version of our architectural blueprint in figure 1.7.

We're going to take on the tiers one at a time, starting from the left with the collection tier. Don't let the lack of emphasis on the message queuing tier in figure 1.7

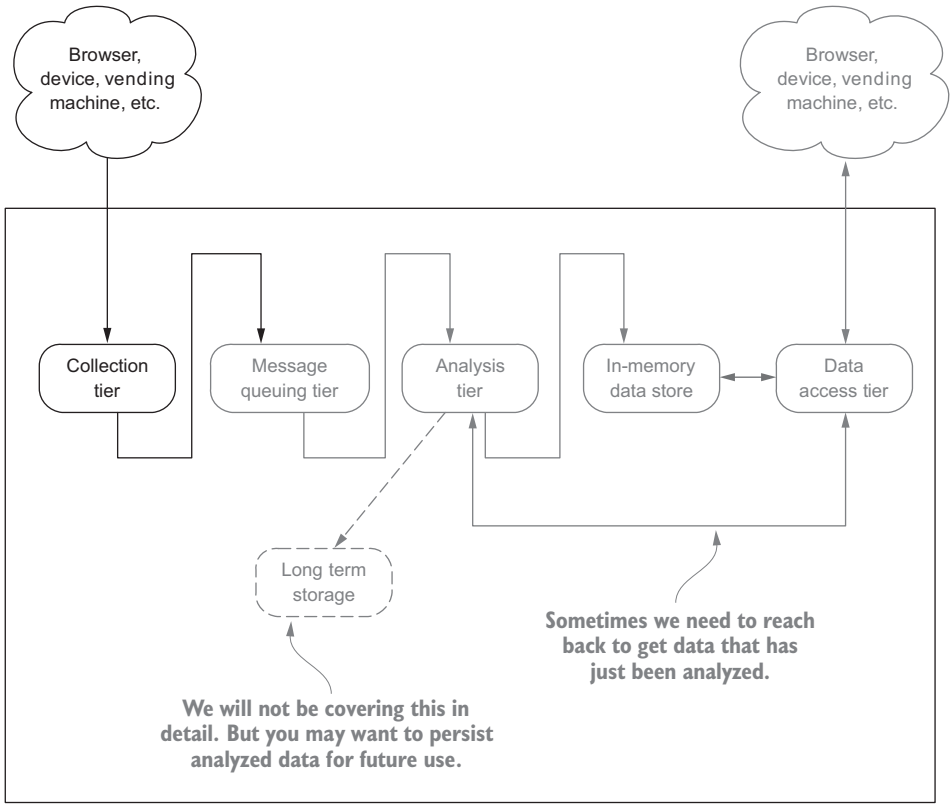

Figure 1.7 Architectural blueprint with emphasis on the first tier

bother you—in certain cases where it serves a collection role, I'll talk about it and clear up any confusion. Now, on to our first tier, the collection tier—our entry point for bringing data into our streaming, in-the-moment system.

2 Getting data from clients: data ingestion

This chapter covers

- Learning about the collection tier
- Understanding the data collection patterns
- Taking the collection tier to the next level
- Protecting against data loss

On to our first tier: the *collection tier* is our entry point for bringing data into our streaming system. Figure 2.1 shows a slightly modified version of our blueprint, with focus on the collection tier.

This tier is where data comes into the system and starts its journey; from here it will progress through the rest of the system. In the coming chapters we'll follow the flow of data through each of the tiers. Your goal for this chapter is to learn about the collection tier. When you finish this chapter you will know about the collection patterns, how to scale, and how to improve the dependability of the tier via the application of fault-tolerance techniques.

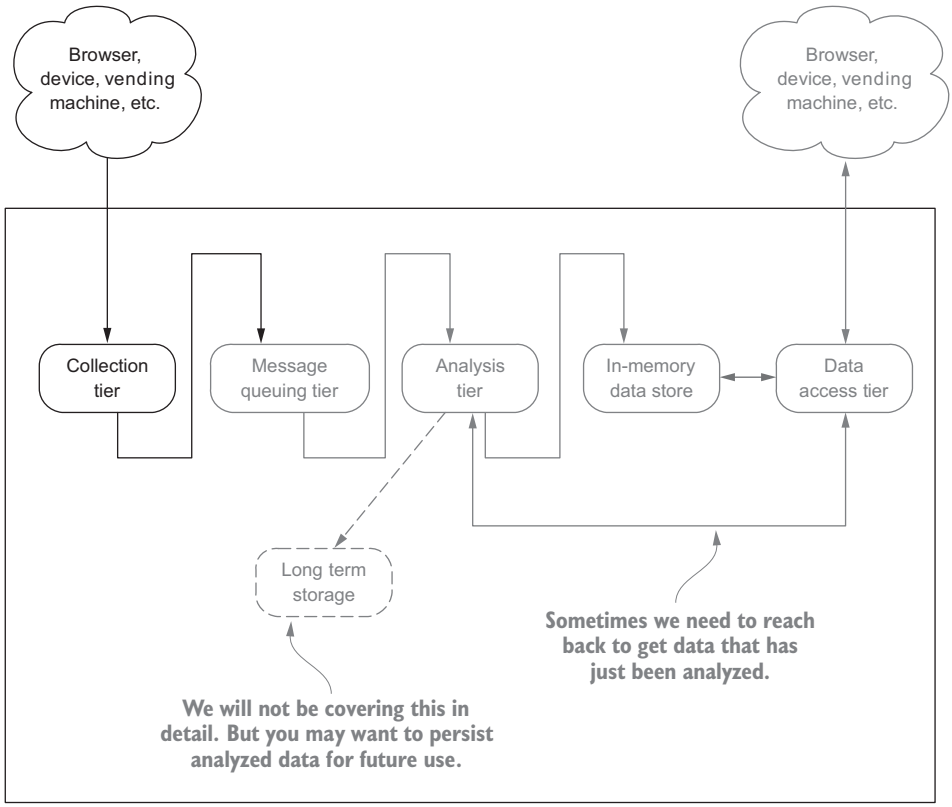

Figure 2.1 Architectural blueprint with emphasis on the collection tier

2.1 Common interaction patterns

Regardless of the protocol used by a client to send data to the collection tier—or in certain cases the collection tier reaching out and pulling in the data—a limited number of interaction patterns are in use today. Even considering the protocols driving the emergence of the Internet of Everything, the interaction patterns fall into one of the following categories:

- Request/response pattern
- Publish/subscribe pattern
- One-way pattern
- Request/acknowledge pattern
- Stream pattern

Let's discuss how you might collect data using each of these patterns.

2.1.1 Request/response pattern

This is the simplest pattern. It's used when the client must have an immediate response or wants the service to complete a task without delay. Every day you experience this pattern while browsing the web, searching for information online, and using your mobile device. Here's how the pattern works. First, a client makes a request to a service—this may be to take an action (such as send a text message, apply for a job, or buy an airline ticket) or to request data (such as perform a search on Google or find the current weather in their city). Second, the service sends a response to the client. Figure 2.2 illustrates this pattern.

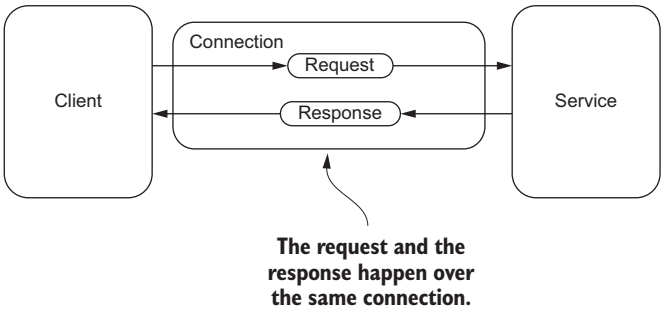

Figure 2.2 Basic request/response pattern

The simplicity of a synchronous request and response pattern comes at the cost of the client having to wait for the response and the service having to respond in a timely fashion. With modern-day services, this cost often results in an unacceptable experience for users. Imagine browsing to your favorite news or social site, and your browser tries to request all the resources in a synchronous fashion. Outside of basic services such as requesting the current weather, the potential delay is no longer tolerable. In many cases this can translate into lost revenue for merchants because users don't want to wait for the response.

Three common strategies can overcome this limitation: one on the client side, one on the service side, and one a combination of the two. Let's consider the client side first. A common strategy often taken by the client is to make the requests asynchronously; this approach is illustrated in figure 2.3.

With this adaptation the client makes the request of the service and then continues on with other processing while the service is processing the request. This is the pattern used by all modern web browsers: the browser makes many asynchronous requests for resources and renders the image and/or content as it arrives. This type of processing allows the client to maximize the time normally spent waiting on the response. The end result is an overall increase in the work performed by the client over a period of time. Implementing this type of pattern is relatively easy today because all modern programming languages and many of the frameworks you may

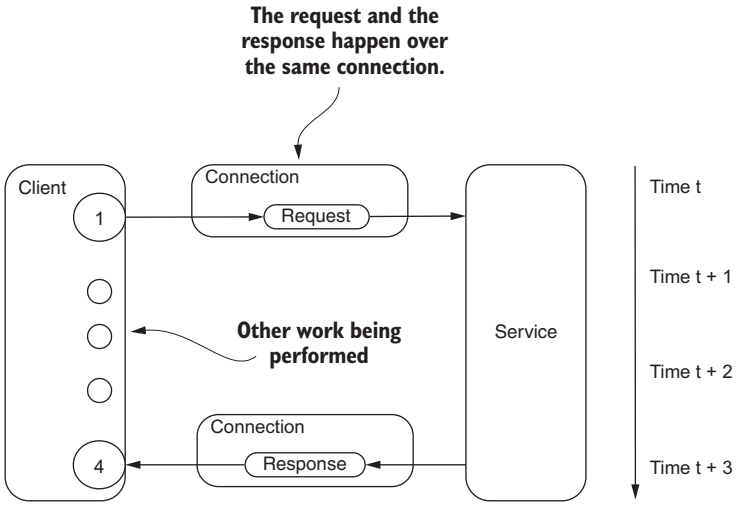

Figure 2.3 Client making asynchronous request to the service

use natively support performing the request asynchronously. This pattern is often called *half-async* because one half of the request response is done asynchronously.

Implementing this type of processing on the service end is also very common and is illustrated in figure 2.4.

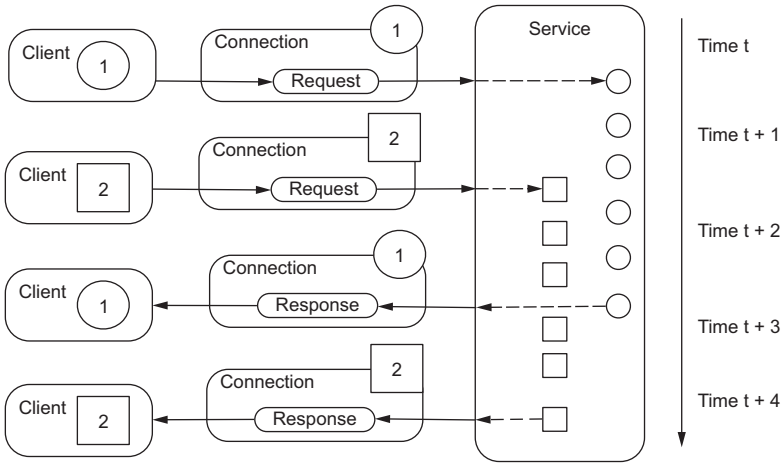

Figure 2.4 Service async request/response pattern

With the service-side half-async pattern, the service receives a request from a client, delegates the work to be done, and when the work is finished responds to the client. This type of processing results in the development of more scalable services, which

can handle requests from many more clients. This type is also very common in all server-side development frameworks found today for all popular programming languages.

The last variation of this pattern, called *full-async*, occurs when both client and server perform their work asynchronously; the resulting flow is the same as that shown in figure 2.4. Today many modern clients and services operate in this full-async fashion.

Now let's walk through an example of using this pattern in the design and development of a streaming system. Imagine we work in the transportation industry, and last week while enjoying coffee with our friend Eric, who works in the automotive industry, we came up with an idea to provide a real-time traffic and routing service for all vehicles on the road. Our company would build the service, and Eric's company would build the streaming system that would reside inside the vehicles. We then sketched out what this solution would look like. Starting with the vehicle part of it, figure 2.5 shows our high-level drawing of the vehicle side of things.

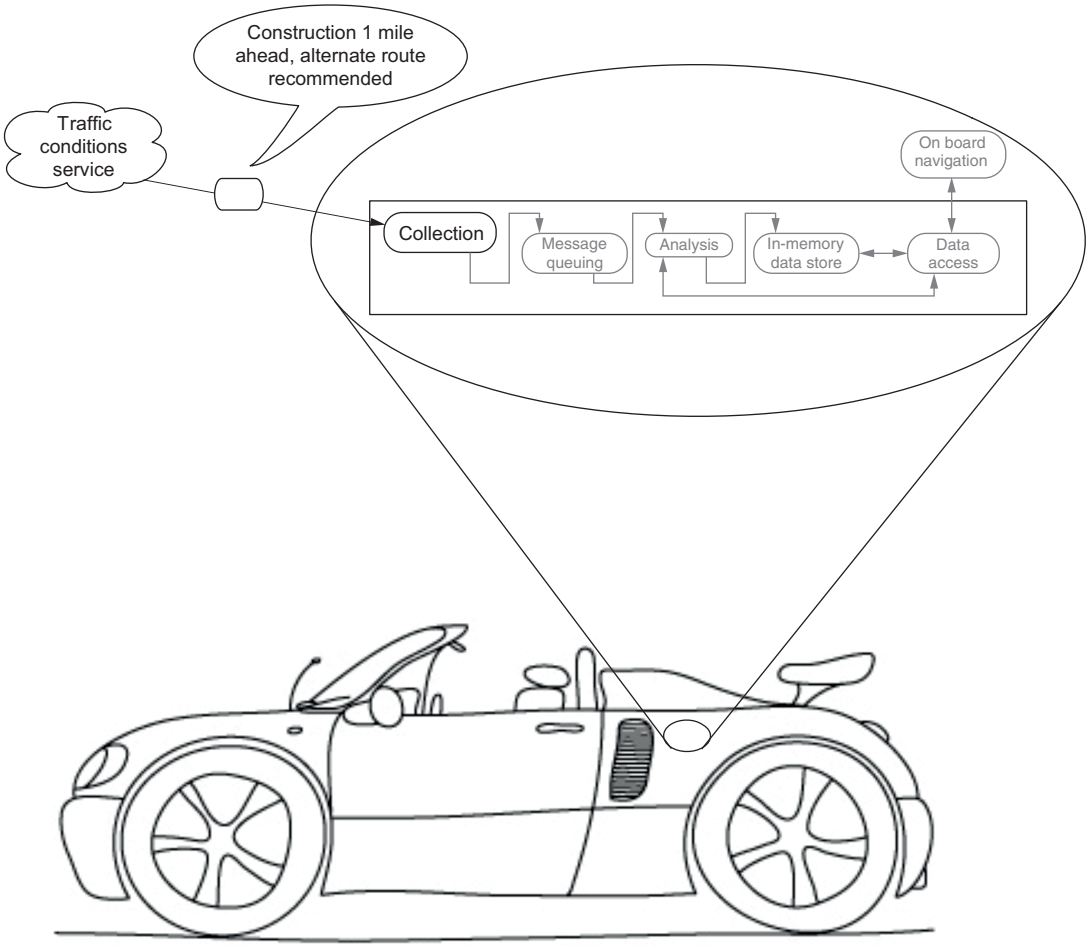

Figure 2.5 Receiving the response to the traffic conditions request with an on-board streaming system

For the vehicle, Eric is going to build an embedded streaming system to not only handle interacting with your traffic and navigation service but also to have the ability to interact with other services and perhaps vehicles.

The request/response pattern would work well for this scenario; in particular we'd want to choose the full-async variant. That way, our traffic and navigation service would be better positioned to handle requests from a lot of cars on the road at a single time. For Eric's on-board streaming system, the ability to asynchronously request data and process it as it arrives would be essential. By following this pattern, the streaming system would not be blocked waiting for a response from our service and could handle other data analysis pertinent to the vehicle.

At this point we're ready for Eric's team to build the vehicle side of things, and we're ready to build the traffic and routing service. If you're interested in learning more about this pattern, a good place to start is with Robert Daigneau's *Service Design Patterns* (Addison-Wesley, 2011).

2.1.2 Request/acknowledge pattern

There are times when you need to use an interaction pattern with similar semantics to the request/response pattern, but you don't need a response from the service. Instead, what you need is an acknowledgment that your request was received. The request/acknowledge pattern fits that need. Often the data sent back in the acknowledgment can be used to make subsequent requests, perhaps to check the status of the initial request or get a final response.

Imagine we're working with the marketing department for our company to make sure that on our e-commerce site we provide the right offer to the right person at the right time, with the goal of increasing their likelihood of making a purchase during their current visit. After further discussions with the marketing team, we settled on a solution that would constantly update a visitor's propensity-to-buy score during their visit. With this dynamic score available, our site can make the right offer at any time to influence their decision to purchase. Figure 2.6 shows how this looks from a high level.

Let's walk through the flow of data illustrated in figure 2.6. As the visitor is browsing our site, we're collecting data about each page they visit and every link they click. The unique thing we're doing that's particular to the request/acknowledge pattern occurs on the first page they visit. On this page our collection tier returns an acknowledgment that can be used in future requests. Unlike the request/response pattern, which may return as response success or failure, the request/acknowledge pattern returns data that can be used in future requests. In this case the acknowledgment is nothing more than a unique identifier, but it plays an important role. The acknowledgment can be used on all subsequent pages the visitor visits. When we call the propensity service, we can pass the unique identifier we obtained on the very first visit. With the unique identifier, which identifies the visitor, our propensity service can determine and return the visitor's current propensity-to-purchase score.

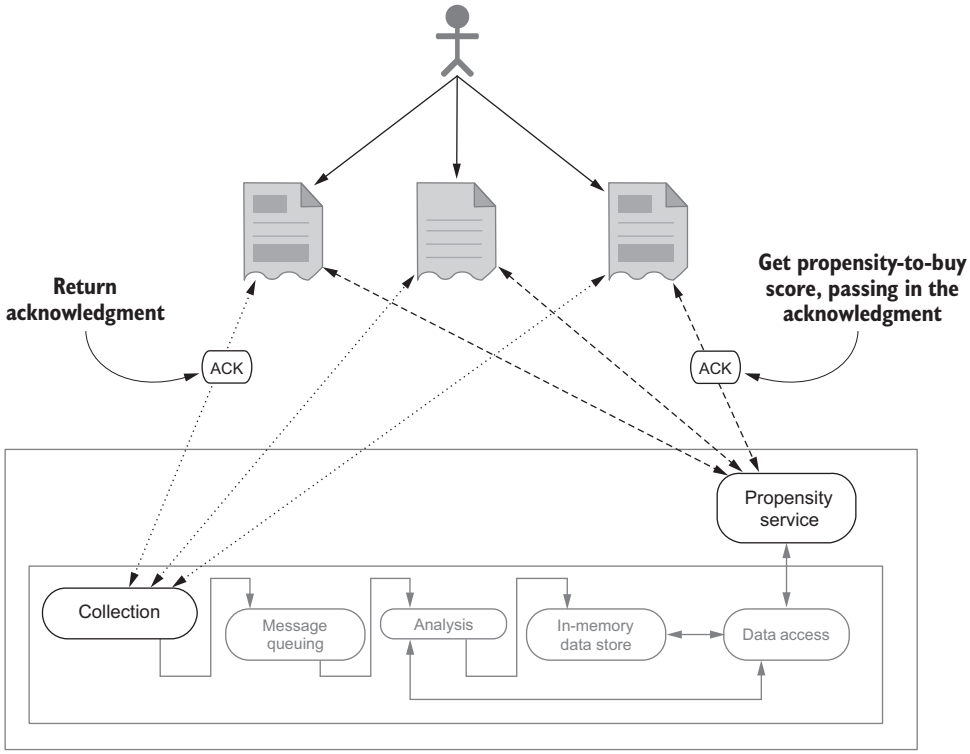

Figure 2.6 Visitor browsing while data is collected and their propensity-to-buy score is updated

I realize I'm leaving out a lot of the details of how we got from collection to a propensity score, but it will become clearer in the coming chapters. The key takeaway is that with the request/acknowledge pattern, a client makes a request of a service, asking for an action to be taken, and in turn receives an acknowledgment token that can be used in future requests. We experience this pattern every day in real life. For example, when you purchase an item online, you're often given a confirmation number, which you can then use to check on the status of your order.

If you're interested in learning more about this pattern, see Gregor Hohpe and Bobby Woolf's *Enterprise Integration Patterns* (Addison-Wesley, 2003).

2.1.3 Publish/subscribe pattern

This is a common pattern with message-based data systems; the general flow is shown in figure 2.7.

The general data flow as illustrated in figure 2.7 starts with a producer publishing a message to a broker. The messages are often sent to a *topic*, which you can think of as a logical grouping for messages. Next, the message is sent to all the consumers

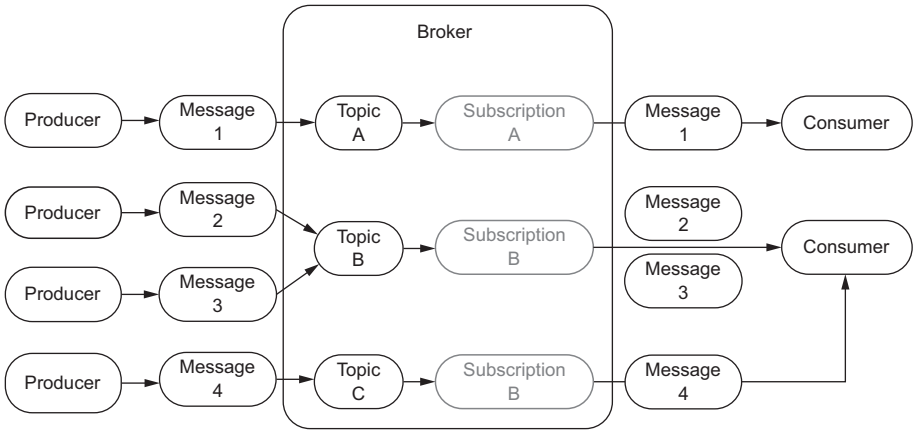

Figure 2.7 General data flow for the publish/subscribe message pattern

subscribing to that topic. There's a subtlety in this last step, covered in-depth in chapter 3. For now, know that some technologies follow the data flow as illustrated here, pushing messages to consumers. But with other technologies, the consumer pulls messages from the brokers. It may not be obvious initially, but often a producer publishing a message doesn't mean that it needs to subscribe to a topic. Nor is it required that a subscriber produce a message.

Let's walk through an example of how this protocol can be used and see its impact on our collection tier. After the success of our joint venture with Eric's company, we started to think about how we can take our in-vehicle traffic and routing service to the next level. After considering several ideas, we settled on the idea of making it social. In addition to the vehicle requesting traffic information and routing, it would send real-time traffic updates back to the service and subscribe to the real-time traffic reports from other vehicles traveling along the same route. Figure 2.8 shows the flow of messages we're talking about.

For simplicity, the figure shows only a handful of cars acting as producers and sending their current traffic conditions to the broker, and a single car acting as the consumer. If this were real, you could imagine how each producer would also consume and analyze all the data. By using the publish/subscribe pattern we're able to decouple the sender of the traffic data from its consumer. As we scale this simple example—four cars sending data and one consuming data—to all cars in the United States, you can imagine how important the decoupling this pattern provides is. If you're interested in learning some of the finer points about this pattern, a good place to start is *Enterprise Integration Patterns*, mentioned previously.

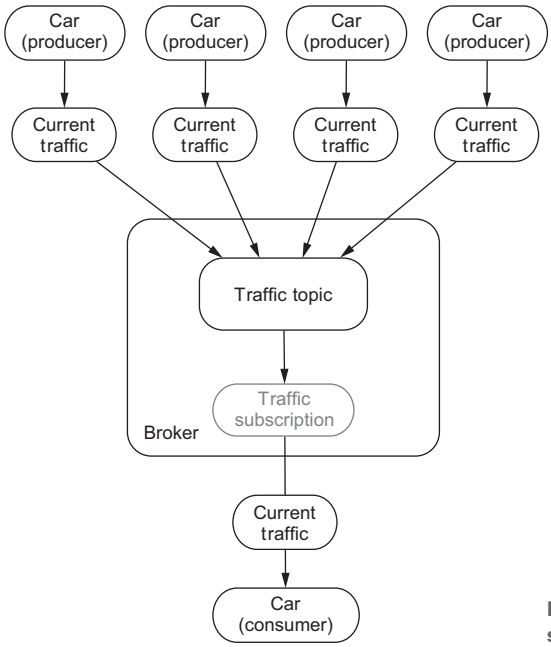

Figure 2.8 Current traffic publish/subscribe message pattern

2.1.4 One-way pattern

This interaction pattern is commonly found in cases where the system making the request doesn't need a response. You may also see this pattern often referred to as the "fire and forget" message pattern. In some cases this pattern has distinct advantages and may be the only way for a client to communicate with a service. It's similar to the request/response and request/acknowledge patterns in the way a message is sent from the client to the service. The major difference is that the service doesn't send back a response. In the other patterns the client knows the request was received and processed; with the one-way pattern the client doesn't even know whether the request was received by the service.

You may be wondering how or where a pattern that has zero guarantees that the message was even received by a service can be useful. It's useful in environments where a client doesn't have the resources or the need to process a request. For example, think of the available data about the servers in your data center. You'd like for the server to send data about how much memory and CPU are being used every 10 seconds. You don't need the server to take any action or even worry about the result; it's purely producing data as fast as possible.

Examples of this interaction pattern appear all around us and will continue to grow along with the proliferation of the Internet of Everything, which is infiltrating many aspects of our life, sports being no exception. A recent partnership between the NFL and Zebra (www.zebra.com/us/en/nfl.html) resulted in players during

Thursday Night Football games being outfitted with quarter-sized radio frequency identifier (RFID) tags on their equipment. Each tag transmits data, such as the athlete's movement, distance, and speed, approximately 25 times a second to the 20 RFID receivers installed in the stadium. Within half a second, the data is analyzed and relayed to the TV broadcast trucks to be used by commentators. In this scenario, the RFID tag, the client, doesn't need and doesn't have the resources to process a response from the RFID receiver.

With the data being sent 25 times a second, if, during one second, five samples were lost and were not received by the RFID receiver, would the resulting analysis be impacted? No, it would not. That's another noteworthy characteristic of this pattern—it is appropriate for and often found in environments where losing some data is tolerable in exchange for simplicity, reduced resource utilization, and speed. To learn more about this pattern, see Nicolai M. Josuttis's *SOA in Practice* (O'Reilly, 2007).

2.1.5 Stream pattern

This interaction style is quite different than all the others discussed so far. With all the other patterns, a client makes a request to a service that may or may not return a response. The stream pattern flips things around, and the service becomes the client. A comparison of this to the other patterns you've seen is illustrated in figure 2.9.

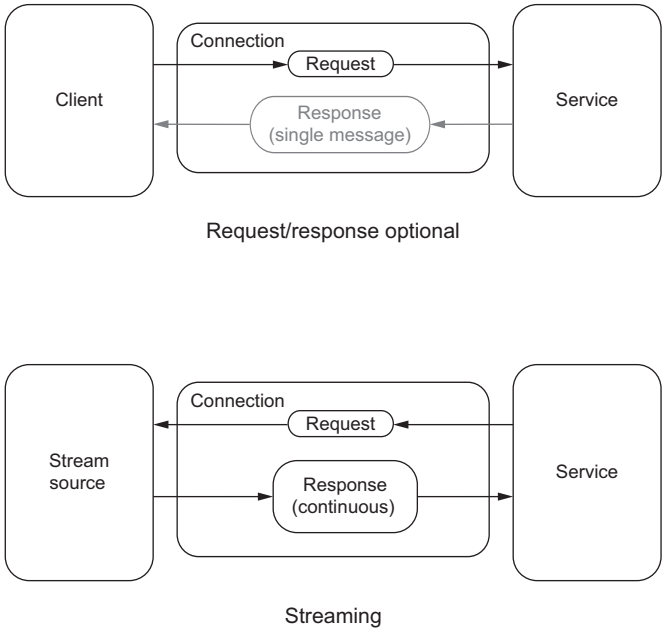

Figure 2.9 Comparing the request/response patterns to the stream pattern

There are a couple of important distinctions to point out when comparing the previous patterns (all the request/response optional patterns) with the stream pattern:

- With the request/response style of interaction as depicted at the top of figure 2.9, the client pushes data to the service in the request, and the service may respond. This response is grayed out in the diagram because the response is not required by some variations of this pattern. It boils down to a single request resulting in zero or one response. The stream pattern as depicted at the bottom of figure 2.9 is quite different; a single request results in no data or a continual flow of data as a response.
- In the request/response optional patterns a client external to the streaming system is pushing the message to it. In our previous examples this was a web browser, a car, or a phone—all clients that send a message to our collection tier. In the case of the stream pattern, our collection tier connects to a stream source and pulls data in. For example, you may be interested in building a streaming system to do sentiment analysis of tweets. To do so, you'd build a collection tier that establishes a connection to Twitter and consumes the stream of tweets.

This pattern is very interesting and powerful: ingesting a stream of data and producing another stream. With this you can quickly build a streaming analysis system that consumes publicly available data and in turn creates new streams of data based on your analysis. Unlike the other patterns, where you need to create or find clients to send a request to your service, with the stream pattern you can chose to connect to and process the data from a stream source.

Meetup.com provides an example input stream that you can use for exploring this interaction pattern or as input to a streaming system. This is also the stream we will use in chapter 9 when we build an end-to-end streaming system. The stream is composed of JSON events, each of which is generated every time someone RSVPs to a meetup. To see this input stream in action, open your favorite browser and go to http://stream.meetup.com/2/rsvps. In this case a simple, long-lived HTTP connection is established, and data is subsequently streamed back to your browser until you end the HTTP connection. In the data stream you'll see JSON events that are similar to the following listing.

See code listings 9.2–9.7

Listing 2.1 Example JSON stream event

```
{
      "venue": {
             "venue_name": "Chicago Symphony Center",
             "lon": -87.624402,
             "lat": 41.878904,
             "venue_id": 700306
   },
   "visibility": "public",
   "response": "yes",
   "member": {
          "member_id": 184005505,
```

Venue-related information

Member-related information

```
            "member_name": "Rifat"
    },
    "rsvp_id": 1631403261,
    "event": {
            "event_name": "Civic Orchestra Open Rehearsal w\/ Riccardo Muti -
     Brahm' s 4th Symphony",
            "event_id": "234044012",
            "time": 1474934400000
},
    "group": {
            "group_topics": [{
                    "urlkey": "symphony",
                    "topic_name": "Symphony"
            }],
            "group_city": "Chicago",
            "group_country": "us",
            "group_id": 882009,
            "group_name": "Chicago Classical Music Events",
            "group_lon": -87.63,
            "group_urlname": "chicagosymphony",
            "group_state": "IL",
            "group_lat": 41.88
     }
}
```

Event-related information

Group-related information

Listing 2.1 is an example of using the stream interaction pattern. Imagine if you took this data and combined it with social data such as tweets about certain events or venues. Again, that's something that's hard to replicate with the other patterns. As you look at this data, I'm sure you'll come up with questions about it without combining it with other streams. Perhaps you want to count the top events people are RSVP'ing to or maybe the top groups by city. But let's not get ahead of ourselves—we're going to work through how to answer these and other questions in chapter 4. For now, it's enough to recognize this pattern and start to get a feel for this interaction pattern.[1]

2.2 Scaling the interaction patterns

Now that we've discussed each of the interaction patterns, let's see how we'd scale our collection tier and talk about some of the things to keep in mind when implementing it. We're going to keep the discussion at the level of the two categories we grouped the interaction patterns into before.

2.2.1 Request/response optional pattern

To discuss scaling this general pattern we'll continue with the example from our initial discussion of the request/response pattern, the real-time traffic and routing service for all vehicles on the road. To get a better sense for the scale of our idea of providing this service for all vehicles on the road in the United States, we'll consider the 2012 National Transportation Statistics report (the last complete year produced by

[1] If you're interested in learning more about this dataset, check out www.meetup.com/meetup_api/docs/stream/2/rsvps/#polling.

the Bureau of Transportation Statistics, www.rita.dot.gov). According to this report, approximately 253 million vehicles were registered in the United States and were driven approximately 2.966 trillion miles in 2012. This means that at any time during the almost 3 trillion miles driven by one of the 253 million vehicles, we may get a request for current traffic conditions and alternate route suggestions. At any moment we'll need to handle thousands and possibly millions of requests.

If you remember, chapter 1 talked about horizontal scaling being our overall goal for every tier of our streaming system. With this example and our use of the request/response optional pattern, horizontal scaling will work very well for two reasons. First, with this pattern we don't have any state information about the client making the request, which means that a client can connect and send a request to any service instance we have running. Second—and this is a result of the stateless nature of this pattern—we can easily add new instances of this service without changing anything about the currently running instances. The mode of scaling stateless services is so popular that many cloud-hosting providers, such as Amazon, offer a feature called auto-scaling that will automatically increase or decrease the number of instances running, based on demand. On top of horizontal scaling, we also want our service to be *stateless*, which will allow any vehicle to make a request to any instance of our service at any time. This stateless trait is commonly found in systems that use this pattern. Taking horizontal scaling and statelessness into consideration, we arrive at figure 2.10, which shows these two aspects together.

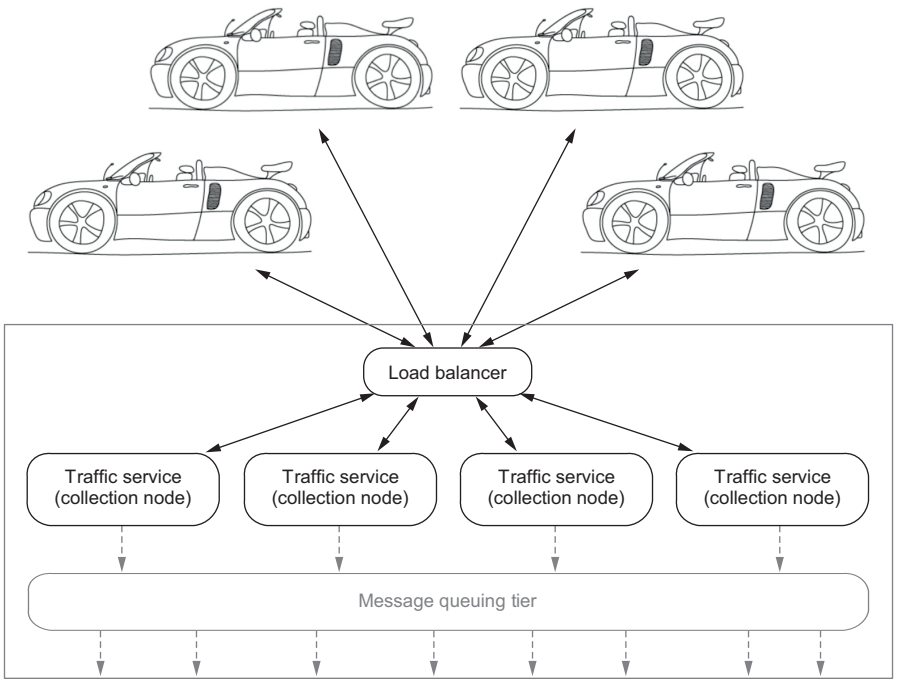

Figure 2.10 Vehicle and traffic service with a load balancer

We're using a load balancer here to be able to route requests from the vehicles to an instance of our service that's running. As instances are started or stopped based on demand, the running instances the load balancer routes requests to will change. We now have a pretty good idea of how we're going to scale our service and the protocol we're going to use with our clients.

2.2.2 Scaling the stream pattern

The meetup.com stream we used as an example in section 2.1.5 for our discussion of the stream interaction pattern has a fairly low velocity (less than 10 events per second). Obviously that's not the best example to help us think through how to scale a collection tier when using the stream interaction pattern. Instead, let's imagine that Google provided a public stream of all the searches being performed as they happen; according to internetlivestats.com (www.internetlivestats.com/one-second/#google-band) at this moment, that would result in approximately 46,000 search events per second. Remember, horizontal scaling is our goal when building each tier of our streaming system. With many streaming protocols, as you saw earlier when you consumed the meetup.com RSVP stream in your browser, there's a direct and persistent connection between the client (our collection tier) and the server (the service we request data from), as illustrated in figure 2.11.

In figure 2.11 you can see that three of the four nodes are idle because there's a direct connection between the search stream and the node handling the stream. To scale our collection tier, we have a couple of options: scale up the collection node

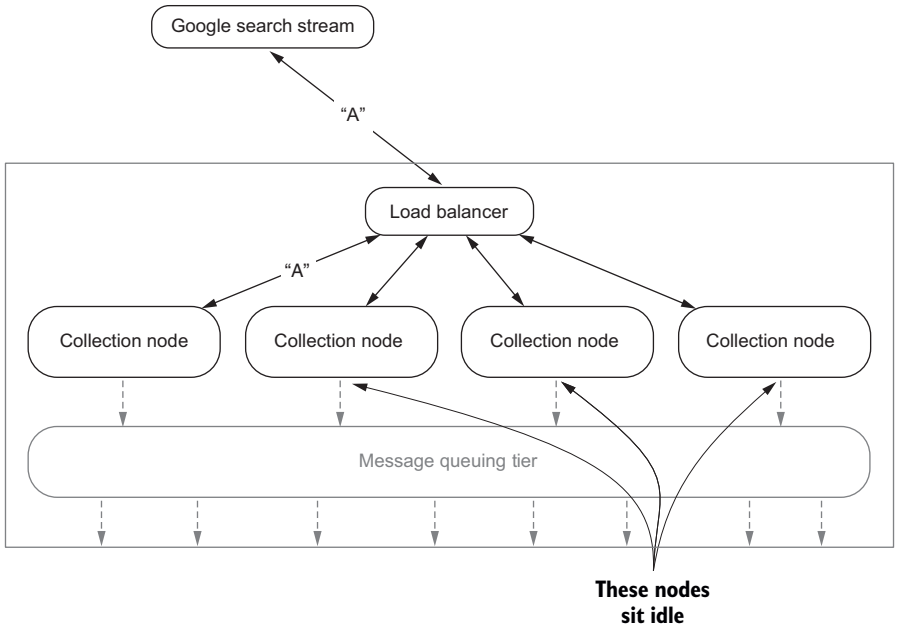

Figure 2.11 Search stream with direct connection to single collection node

that's consuming the stream, and introduce a buffering layer in the collection tier. These are not mutually exclusive, and depending on the volume and velocity of the stream, both may be required. Scaling up the node consuming the stream will get us only so far; at a certain point we'll reach the limits of the hardware our collection node is running on and won't be able to scale it up any further. Figure 2.12 shows what our collection tier looks like with the buffering layer in place.

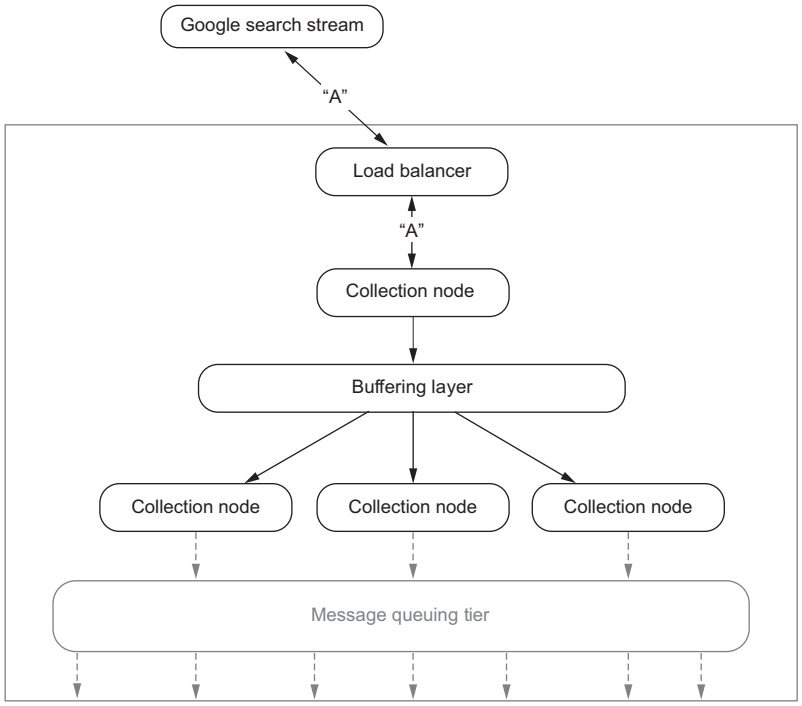

Figure 2.12 Collection tier with buffering layer in place

The key to being able to put a buffering layer in the middle lies in making sure no business logic is performed on the messages when they're consumed from the stream. Instead, they should be consumed from the stream and as quickly as possible pushed to the buffering layer. A separate set of collection nodes that may perform business logic on the messages will then consume the messages from the buffering layer. The second set of collection nodes can now also be scaled horizontally.

2.3 Fault tolerance

Regardless of the interaction pattern used, one thing is for sure: at some point one or more of our collection nodes will fail. The failure may be the result of a bug in our software, third-party software we rely on, or the hardware our service runs on. Regardless of the cause, our goal is to mask the failures and to improve the dependability of

our collection tier. You may be wondering why we need to worry about this if we've done our job of horizontally scaling and increasing the redundancy of our tier. That's a fair question. The answer is quite simple: the message our collection tier receives from a client may not be reproducible. In essence, there may be no way for our collection tier to ask for the client to send us the data again, and in many cases no way for the client to do so even if our collection tier could ask.

Depending on your business, there may be times when it's okay to lose data, but in many cases it's not okay. This section explores the fault-tolerance techniques you can employ to ensure that we don't lose data and we improve the dependability of our collection tier. Our overarching goal is that when a collection node crashes (and it will), we don't lose data and can recover as if the crash had never occurred. To understand the areas we need to protect, look at figure 2.13, which shows the simplest possible collection scenario with the places we can lose data when the node crashes.

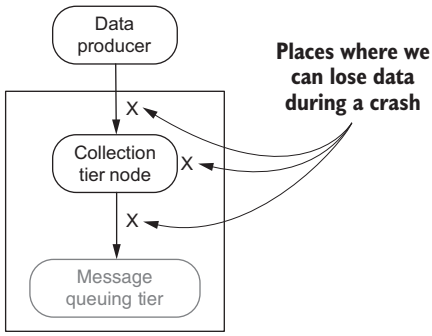

Figure 2.13 Collection scenario with our data-loss potential identified

The two primary approaches to implementing fault tolerance, *checkpointing* and *logging*, are designed to protect against data loss and enable speedy recovery of the crashed node. The characteristics of checkpointing and logging are not the same, as you'll soon see.

First, let's consider checkpointing. Numerous checkpoint-based protocols are available to choose from in the literature, but when you boil it down, the following two characteristics can be found in all of them:

- *Global snapshot*—The protocols require that a snapshot of the global state of the whole system be regularly saved to storage somewhere, not merely the state of the collection tier.
- *Potential for data loss*—The protocols only ensure that the system is recoverable up to the most recent recorded global state; any messages that were processed and generated afterward are lost.

What does it mean to have a global snapshot? It means we're able to capture the entire state of all data and computations from the collection tier through the data access tier and save it to a durable persistent store. That's what I'm talking about when

I refer to the *global state* of the system. This state is then used during recovery to put the system back into the last known state. The potential for data loss exists if we can't capture the global state every time data is changed in the system.

An example I'm sure you've seen before is autosave in popular document-editing software such as Microsoft Word or Google Docs. A snapshot is taken of the document as you are editing it, and if the application crashes, you can recover to the last checkpoint. If you're like many people, you've seen checkpointing and potential data loss in action when your word processing program crashed and your most recent edits were not saved.

When considering using a checkpoint protocol for implementing fault tolerance in a streaming system, keep two things in mind: the implications of the previously mentioned attributes and the fact that a streaming system is composed of many layers and many different technologies. This layering and the data movement make it very hard to consistently capture a global snapshot at a point in time, and that makes checkpointing a bad choice for a streaming system. But checkpointing is a valid choice if you're building the next version of HDFS or perhaps a new NoSQL data store. Given that checkpointing isn't a good match for a streaming system, we won't spend more time on these protocols. Even though they're not a good fit, they're fascinating to study. If you're interested in learning about them, I would encourage you to start with the great article by Elnozahy, En Mootaz, et al., "A Survey of Rollback-Recovery Protocols in Message-Passing Systems" (*ACM Computing Surveys* 34.3 (2002): 375–408).[2]

Turning our attention to the logging protocols, you have a variety to choose from. Reduced to their essence, you'll find that they all share the common goals of overcoming the expense and complexity of checkpointing and providing the ability to recover up to the last message received before a crash. Part of the complexity of checkpointing that's eliminated is the global snapshot, and therefore the management and generation of the global state. In the end, the goals of the logging technique manifest themselves in the basic idea that underpins all of the logging techniques: *if a message can be replayed, then the system can reach a global consistent state without the need for a global snapshot.*

This means that each tier in the system independently records all messages it receives and plays them back after a crash. Implementing a logging protocol frees us from worrying about maintaining global state, enabling us to focus on how to add fault tolerance to the collection tier. To do this we're going to discuss two classic techniques, *receiver-based message logging* (RBML) and *sender-based message logging* (SBML), and an emerging technique called *hybrid message logging* (HML). Along the way we'll also discuss how and why we can use these with our collection tier.

Before moving on to discuss these techniques, look at figure 2.14, which illustrates how they fit together and what data we're trying to protect.

Figure 2.14 shows a single collection tier node that's receiving a message, performing some logic on it, and then sending it to the next tier. As their names imply, receiver-based

[2] The article can be downloaded from http://mng.bz/vUz2.

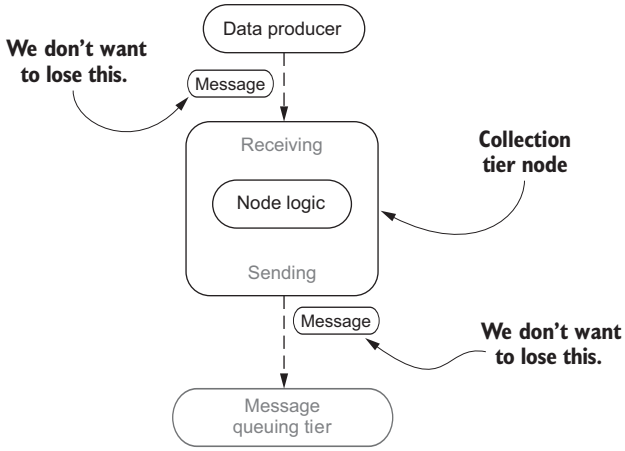

Figure 2.14 High-level overview of receiver-based and sender-based message logging

logging is concerned with protecting the data the node is receiving, and sender-based logging is concerned with protecting the data that's going to be sent to the next tier. Imagine your business logic being sandwiched between two layers of logging, one designed to capture the data before it's changed and one to capture it before it's sent to the next tier. If you're thinking that this is a lot of potential overhead and overlap, in some cases it may be, and this is where HML aims to strike a balance between RBML and SBML. With that frame of reference, let's start our discussion with RBML.

2.3.1 Receiver-based message logging

The RBML technique involves synchronously writing every received message to stable storage before any action is taken on it. By doing that, we can ensure that when our software crashes while handling the message, we already have it saved and upon recovering we can replay the message. Figure 2.15 illustrates how our collection node changes with the introduction of RBML.

In figure 2.15 the message flows from step 1 to step 5; this shows the happy path when there is no failure. We'll walk through the recovery side of it shortly, but first let's review the flow:

1. A message is sent from a data producer (any client).
2. A new piece of software we wrote for the collection node, called the RBML logger, gets the message from the data producer and sends it to storage.
3. The message is written to stable storage.
4. The message then proceeds through to any other logic we have in the node; perhaps we want to enrich the data we're collecting, filter it, and/or route it based on business rules. The important thing is we are recording the data as soon as it is received and before we do anything to it.
5. The message is then sent to the message queuing tier, the next tier in the streaming system.

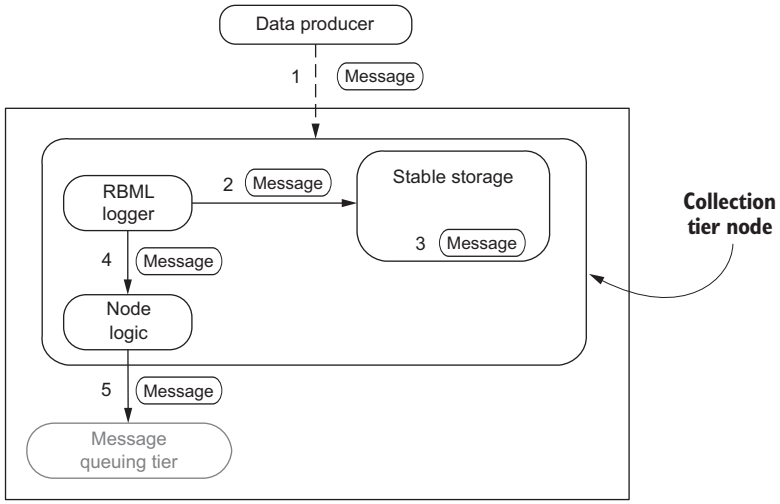

Figure 2.15 RBML implemented for the simple collection node showing the happy path

It's important to point out that depending on the type of stable storage used, steps 2 and 3 may negatively impact the throughput performance of our collection node, sometimes noted as one of the drawbacks to logging protocols. The hybrid message logging technique discussed in section 2.3.3 helps address some of those concerns. For now, we'll keep it simple—at the end of the day the simplicity and recoverability of using RBML for our collection node wins.

Now that you understand how the data flows during normal operation, look at figure 2.16, which shows what the recovery data flow looks like.

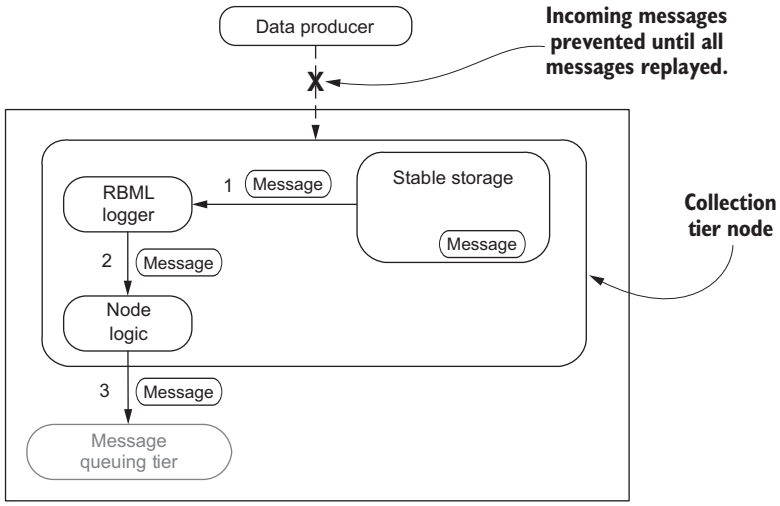

Figure 2.16 The recovery data flow for RBML

In figure 2.16 there are a couple of things to call out. First, once the crash occurs, all incoming messages to this collection node are stopped. Because you'll have more than one collection node and they'll be behind a load balancer, you would take this node out of rotation. Next, the RBML logger reads the messages that have not been processed from stable storage and sends them through the rest of the node logic as if nothing has happened. Lastly, after all pending messages are processed, the node is considered restored and can be put back into rotation, and the data flow resumes, as in figure 2.15.

2.3.2 Sender-based message logging

The SBML technique involves writing the message to stable storage before it is sent. If the RBML technique logs all messages that come in the front door of our collection node to protect us from ourselves, then SBML is the act of logging all outgoing messages from our collection node before we send them, protecting ourselves from the next tier crashing or a network interruption. Figure 2.17 shows the data flow for SBML.

Now that you understand RBML and in particular the data flow, I suspect that the data flow for SBML as depicted in figure 2.17 seems fairly reasonable to you through step 5. One difference is that with RBML we are recording the message as soon as it is received before we do anything to it, and with SBML we are recording it before we send any data to the next tier. The data recorded by an RBML logger is the raw incoming data, and the data recorded by the SBML logger is after our node logic executes (remember, we may have augmented the data in some way) and before we send it on.

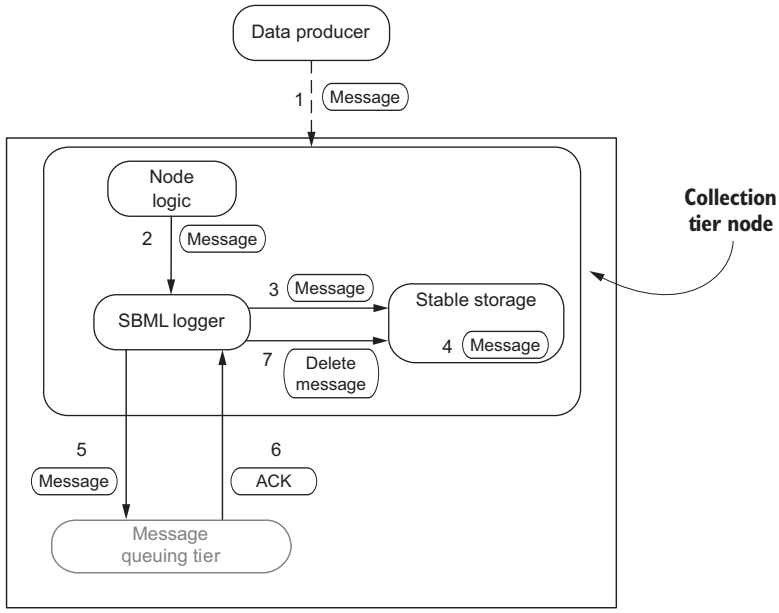

Figure 2.17 The normal execution data flow for SBML

Besides this nuance there's a little wrinkle we'll need to deal with during recovery. During recovery, how do we know whether the next tier has already processed the message we're replaying? There are several ways to handle this. One, shown in figure 2.17, is that we use a message queuing tier that returns an acknowledgment that it received the message. With that acknowledgment in hand, we can either mark the message as replayed in stable storage or delete it from stable storage because we no longer need to replay it during recovery. If the technology you choose for your message queuing tier doesn't support returning an acknowledgment of any sort, then you may be forced into a situation where if no error occurs when sending the message to the message queuing tier, steps 6 and 7 will result in you deleting the message from stable storage.

The recovery data flow as illustrated in figure 2.18 is a little more complex than how we handled recovery with the RBML.

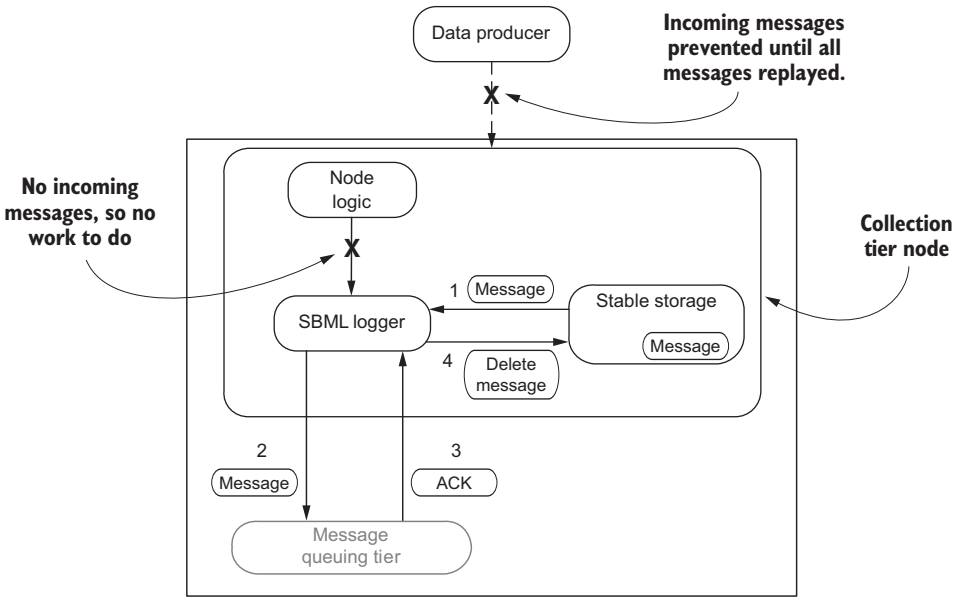

Figure 2.18 Recovery data flow for SBML

I think you'll agree that the recovery data flow for SBML is only marginally more complex than the RBML workflow, but it shouldn't look too foreign to you.

2.3.3 Hybrid message logging

See code listings 9.1 and 9.8

If we stopped right now, we'd have two solutions that we can put in place to address our data loss concerns and dependability: RBML to handle incoming messages and SBML to handle outgoing messages. As mentioned, writing to stable storage can negatively impact our collection node's performance. Implementing both RBML and SBML

means we're writing to stable storage at least twice during normal execution. Some may argue that we'll be doing more logging than processing of data; figure 2.19 suggests they may not be far off.

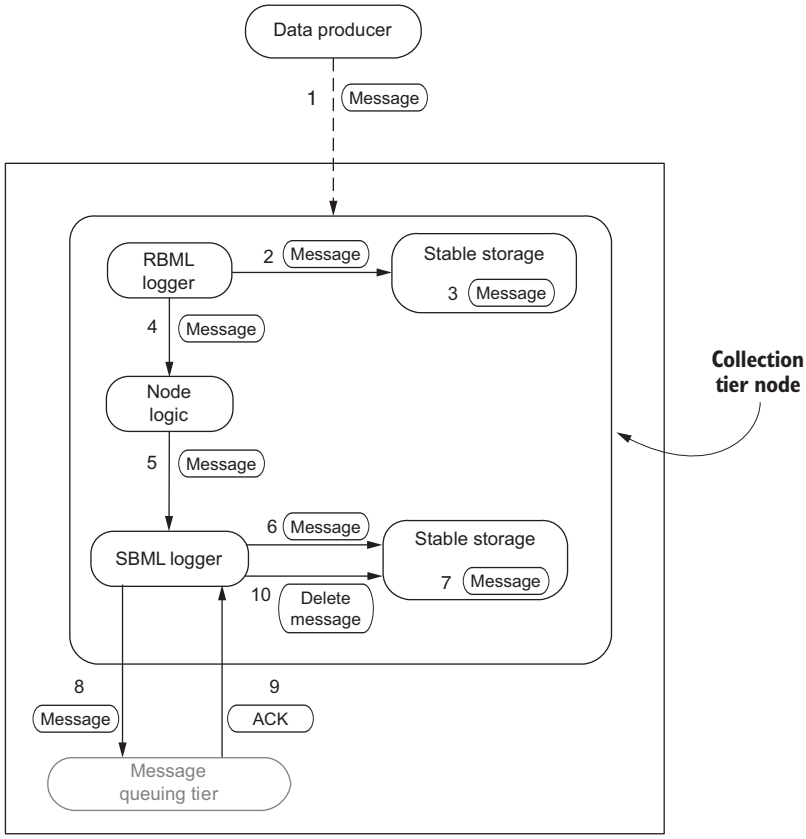

Figure 2.19 RBML and SBML together in the collection tier node

To help with this, hybrid message logging was designed to reduce the impact of message logging during normal processing. To accomplish this, the best parts of RBML and SBML are used at the cost of minimal additional complexity. HML is also designed to provide the same data-loss protection and recoverability found in RBML, SBML, and other logging techniques. There are several ways to implement HML; one common approach is illustrated in figure 2.20.

It's apparent when comparing the data flow in figure 2.19 with both RBML and SBML to the HML data flow in figure 2.20 that the HML approach is slightly less complex. Several factors contribute to this simplification. The first one, which may not come as a surprise, is that the two stable storage instances have been consolidated. This is a minor change, but it allows you to reduce the number of moving parts. The

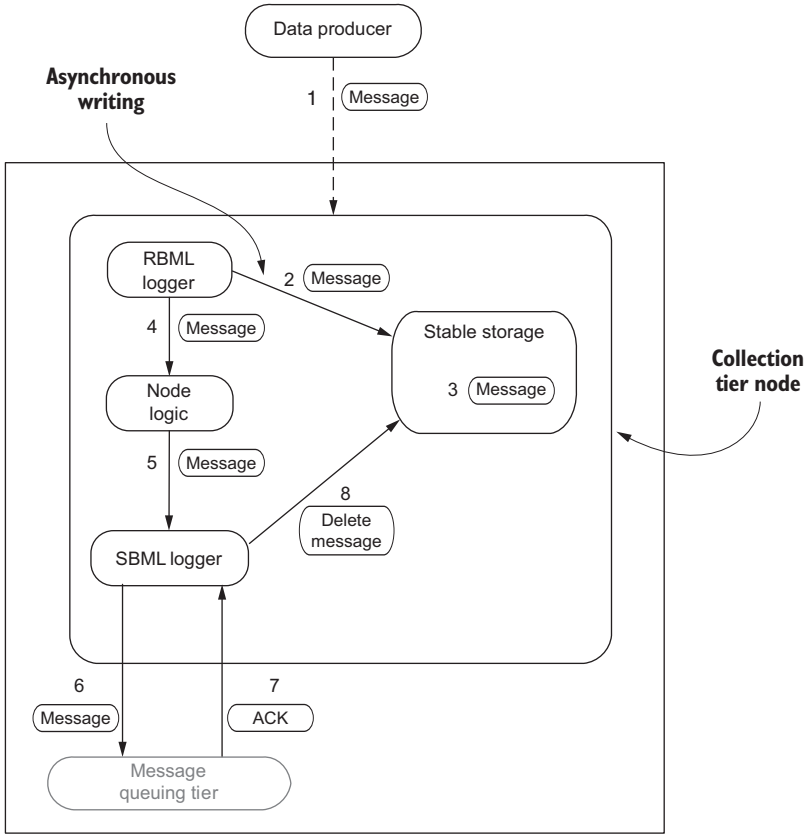

Figure 2.20 HML sample data flow

second change, writing to stable storage asynchronously, has a subtle difference. Arguably this has a more profound impact on the implementation complexity and performance. The complexity comes from making sure you are correctly handling any errors that happen, and the performance comes from using the multi-core world we live in to perform more than one task at a time.

The rest of the data flow should be routine for you by now. If you feel comfortable with the additional complexity and have a choice of implementing HML or standard RBML and SBML, you should implement HML because it will reduce the performance impact of message logging while still providing all the fault tolerance and safety of RBML and SBML. If you're interested in learning more about HML, a great place to start is an article on hybrid message logging by Hugo Meyer and others.[3] In

[3] H. Meyer, Rexachs, D., and Luque, E., "Hybrid Message Logging. Combining Advantages of Sender-based and Receiver-based Approaches" (*Procedia Computer Science* 29 (2014): 2380–90), http://mng.bz/5ZUc.

their research, they showed how they were able to achieve a 43% overhead reduction over the RBML approach.

2.4 A dose of reality

Here's a funny little story to put some of this scaling and fault tolerance into perspective. One time I was working on a streaming system that was populating fancy dashboards for marketers. It had all the bells and whistles—scaling, fault tolerance, monitoring, alerting—the whole nine yards. We had to have all of this and could not lose any data, because our customers wouldn't accept a solution that didn't have complete data. Once this system was running in production, I was curious as to how well our web-based dashboards that consumed our stream via WebSockets were keeping up. Well, come to find out, many of our customers were only able to keep up with about 60% of the stream that was being sent to them; the other 40% of the data was being dropped because they couldn't read it fast enough. When I mentioned this to coworkers, they were shocked and somewhat in disbelief because our customers and business folks loved what they were seeing.

It put things in perspective: the dashboards we produced were showing a picture of our customers' business that was not distorted by the missing data. To me, this was like the difference between high-end HDTV and mid-level HDTV—sure, the quality of the picture may be slightly better, but the picture doesn't change. I'm not implying that you don't need to worry about scaling or fault tolerance, but it's good to keep things in perspective and then reflect on the difference between "we must have *xyz* features" and reality.

2.5 Summary

We've covered a lot of ground in this chapter, exploring the various aspects of collecting data for a streaming system from interaction patterns through scaling and fault-tolerance techniques.

Along the way you

- Learned about the collection tier
- Developed an understanding of various collection patterns
- Had a chance to interact with a live stream
- Learned how to think about scaling your collection tier
- Learned about common fault-tolerance techniques