SAMPLE CHAPTER

RabbitMQ IN DEPTH

Gavin M. Roy

MANNING

RabbitMQ in Depth

by Gavin Roy

Chapter 1

brief contents

1 Foundational RabbitMQ

This chapter covers

- Unique features of RabbitMQ
- Why RabbitMQ is becoming a popular choice for the centerpiece of messaging-based architectures
- The basics of the Advanced Messaging Queuing model, RabbitMQ's foundation

Whether your application is in the cloud or in your own data center, RabbitMQ is a lightweight and extremely powerful tool for creating distributed software architectures that range from the very simple to the incredibly complex. In this chapter you'll learn how RabbitMQ, as messaging-oriented middleware, allows tremendous flexibility in how you approach and solve problems. You'll learn how some companies are using it and about key features that make RabbitMQ one of the most popular message brokers today.

1.1 RabbitMQ's features and benefits

RabbitMQ has many features and benefits, the most important of which are

- *Open source*—Originally developed in a partnership between LShift, LTD, and Cohesive FT as RabbitMQ Technologies, RabbitMQ is now owned by Pivotal Software Inc. and is released under the Mozilla Public License. As an open-source project written in Erlang, RabbitMQ enjoys freedom and flexibility, while leveraging the strength of Pivotal standing behind it as a product. Developers and engineers in the RabbitMQ community are able to contribute enhancements and add-ons, and Pivotal is able to offer commercial support and a stable home for ongoing product maturation.
- *Platform and vendor neutral*—As a message broker that implements the platform- and vendor-neutral Advanced Message Queuing Protocol (AMQP) specification, there are clients available for almost any programming language and on all major computer platforms.
- *Lightweight*—It is lightweight, requiring less than 40 MB of RAM to run the core RabbitMQ application along with plugins, such as the Management UI. Note that adding messages to queues can and will increase its memory usage.
- *Client libraries for most modern languages*—With client libraries targeting most modern programming languages on multiple platforms, RabbitMQ makes a compelling broker to program for. There's no vendor or language lock-in when choosing how you'll write programs that will talk to RabbitMQ. In fact, it's not uncommon to see RabbitMQ used as the centerpiece between applications written in different languages. RabbitMQ provides a useful bridge that allows for languages such as Java, Ruby, Python, PHP, JavaScript, and C# to share data across operating systems and environments.
- *Flexibility in controlling messaging trade-offs*—RabbitMQ provides flexibility in controlling the trade-offs of reliable messaging with message throughput and performance. Because it's not a "one size fits all" type of application, messages can designate whether they should be persisted to disk prior to delivery, and, if set up in a cluster, queues can be set to be highly available, spanning multiple servers to ensure that messages aren't lost in case of server failure.
- *Plugins for higher-latency environments*—Because not all network topologies and architectures are the same, RabbitMQ provides for messaging in low-latency environments and plugins for higher-latency environments, such as the internet. This allows for RabbitMQ to be clustered on the same local network and share federated messages across multiple data centers.
- *Third-party plugins*—As a center point for application integrations, RabbitMQ provides a flexible plugin system. For example, there are third-party plugins for storing messages directly into databases, using RabbitMQ directly for database writes.

- *Layers of security*—In RabbitMQ, security is provided in multiple layers. Client connections can be secured by enforcing SSL-only communication and client certificate validation. User access can be managed at the virtual-host level, providing isolation of messages and resources at a high level. In addition, access to configuration capabilities, reading from queues, and writing to exchanges is managed by regular expression (regex) pattern matching. Finally, plugins can be used for integration into external authentication systems like LDAP.

We'll explore the features on this list in later chapters, but I'd like to focus right now on the two most foundational features of RabbitMQ: the language it's programmed in (Erlang), and the model it's based on (the Advanced Message Queuing model), a specification that defines much of the RabbitMQ lexicon and its behavior.

1.1.1 RabbitMQ and Erlang

As a highly performant, stable, and clusterable message broker, it's no surprise that RabbitMQ has found a home in such mission-critical environments as the centerpiece of large-scale messaging architectures. It was written in Erlang, the telco-grade, functional programming language designed at the Ericsson Computer Science Laboratory in the mid-to-late 1980s. Erlang was designed to be a distributed, fault-tolerant, soft real-time system for applications that require 99.999% uptime. As a language and runtime system, Erlang focuses on lightweight processes that pass messages between each other, providing a high level of concurrency with no shared state.

> **REAL-TIME SYSTEM** A real-time system is a hardware platform, software platform, or combination of both that has requirements defined for when it must return a response from an event. A soft real-time system will sacrifice less important deadlines for executing tasks in favor of more important ones.

Erlang's design, which focused on concurrent processing and message passing, made it a natural choice for a message broker like RabbitMQ: As an application, a message broker maintains concurrent connections, routes messages, and manages their states. In addition, Erlang's distributed communication architecture makes it a natural for RabbitMQ's clustering mechanism. Servers in a RabbitMQ cluster make use of Erlang's *inter-process communication* (IPC) system, offloading the functionality that many competing message brokers have to implement to add clustering capabilities (figure 1.1).

Despite the advantages RabbitMQ gains by using Erlang, the Erlang environment can be a stumbling block. It may be helpful to learn some Erlang so you're confident in managing RabbitMQ's configuration files and using Erlang to gather information about RabbitMQ's current runtime state.

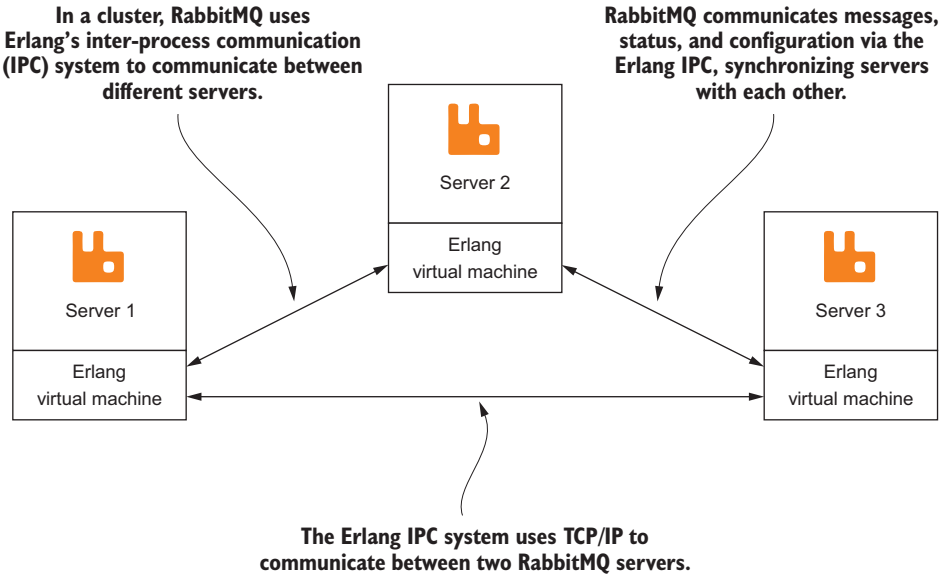

Figure 1.1 RabbitMQ clusters use the native Erlang inter-process communication mechanism in the VM for cross-node communication, sharing state information and allowing for messages to be published and consumed across the entire cluster.

1.1.2 RabbitMQ and AMQP

RabbitMQ was originally released in 2007, and interoperability, performance, and stability were the primary goals in mind during its development. RabbitMQ was one of the first message brokers to implement the AMQP specification. By all appearances, it set out to be the reference implementation. Split into two parts, the AMQP specification defines not only the wire protocol for talking to RabbitMQ, but also the logical model that outlines RabbitMQ's core functionality.

> **NOTE** There are multiple versions of the AMQP specification. For the purposes of this book, we'll focus only on AMQP 0-9-1. Although newer versions of RabbitMQ support AMQP 1.0 as a plugin extension, the core RabbitMQ architecture is more closely related to AMQP 0-8 and 0-9-1. The AMQP specification is primarily comprised of two documents: a top-level document that describes both the AMQ model and the AMQ protocol, and a more detailed document that provides varying levels of information about every class, method, property, and field. More information about AMQP, including the specification documents, may be found at http://www.amqp.org.

There are multiple popular message brokers and messaging protocols, and it's important that you consider the impact that the protocol and broker will have on your application. RabbitMQ supports AMQP, but it also supports other protocols, such as MQTT,

Stomp, and XMPP. RabbitMQ's protocol neutrality and plugin extensibility make it a good choice for multiprotocol application architectures when compared to other popular message brokers.

It's RabbitMQ's roots in the AMQP specification that outline its primary architecture and communication methodologies. This is an important distinction when evaluating RabbitMQ against other message brokers. As with AMQP, RabbitMQ set out to be a vendor-neutral, platform-independent solution for the complex needs that messaging oriented architectures demand, such as flexible message routing, configurable message durability, and inter-datacenter communication, to name a few.

1.2 Who's using RabbitMQ, and how?

As an open-source software package, RabbitMQ is rapidly gaining mainstream adoption, and it powers some of the largest, most trafficked websites on the internet. Today, RabbitMQ is known to run in many different environments and at many different types of companies and organizations:

- Reddit, the popular online community, uses RabbitMQ heavily in the core of their application platform, which serves billions of web pages per month. When a user registers on the site, submits a news post, or votes on a link, a message is published into RabbitMQ for asynchronous processing by consumer applications.
- NASA chose RabbitMQ to be the message broker for their Nebula platform, a centralized server management platform for their server infrastructure, which grew into the OpenStack platform, a very popular software platform for building private and public cloud services.
- RabbitMQ sits at the core of Agoura Games' community-oriented online gaming platform, and it routes large volumes of real-time single and multiplayer game data and events.
- For the Ocean Observations Initiative, RabbitMQ routes mission-critical physical, chemical, geological, and biological data to a distributed network of research computers. The data, collected from sensors in the Southern, Pacific, and Atlantic Oceans, is integral to a National Science Foundation project that involves building a large-scale network of sensors in the ocean and seafloor.
- Rapportive, a Gmail add-on that places detailed contact information right inside the inbox, uses RabbitMQ as the glue for its data processing systems. Billions of messages pass through RabbitMQ monthly to provide data to Rapportive's web-crawling engine and analytics system and to offload long-running operations from its web servers.
- MercadoLibre, the largest e-commerce ecosystem in Latin America, uses RabbitMQ at the heart of their Enterprise Service Bus (ESB) architecture, decoupling their data from tightly coupled applications, allowing for flexible integrations with various components in their application architecture.

- Google's AdMob mobile advertising network uses RabbitMQ at the core of their RockSteady project to do real-time metrics analysis and fault-detection by funneling a fire hose of messages through RabbitMQ into Esper, the complex-event-processing system.
- India's biometric database system, Aandhaar leverages RabbitMQ to process data at various stages in its workflow, delivering data to their monitoring tools, data warehouse, and Hadoop-based data processing system. Aandhaar is designed to provide an online portable identity system for every single resident of India, covering 1.2 billion people.

As you can see, RabbitMQ isn't only used by some of the largest sites on the internet, it's also found its way into academia for large-scale scientific research, and NASA found it fitting to use RabbitMQ at the core of their network infrastructure management stack. As these examples show, RabbitMQ has been used in mission-critical applications in many different environments and industries with tremendous success.

1.3 The advantages of loosely coupled architectures

When I first started to implement a messaging based architecture, I was looking for a way to decouple database updates related to when a member logged in to a website. The website had grown very quickly, and due to the way we'd written it, it wasn't initially designed to scale well. When a user logged in to the website, several database servers had tables that needed to be updated with a login timestamp (figure 1.2). This timestamp needed to be updated in real time, as the most engaging activities on the site were driven in part by the timestamp value. Upon login, members were given preferential status in social games compared to those users who were actively online at any given time.

As the site continued to grow, the amount of time it took for a member to log in also grew. The reason for this was fairly straightforward: When adding a new application that used the member's last login timestamp, its database tables would carry the value to make it as fast as possible by removing cross database joins. To keep the data up to date and accurate, the new data tables would also be updated when the member logged in. It wasn't long before there were quite a few tables that were being maintained this way. The performance issue began to creep up because the database updates were being performed serially. Each query updating the member's last login timestamp would have to finish before the next began. Ten queries that were considered performant, each finishing within 50 ms, would add up to half a second in database updates alone. All of these queries would have to finish prior to sending the authorization response and redirect back to the user. In addition, any operational issues on a database server compounded the problem. If one database server started responding slowly or became unresponsive, members could no longer log in to the site.

To decouple the user-facing login application from directly writing to the database, I looked into publishing messages to message-oriented middleware or a centralized

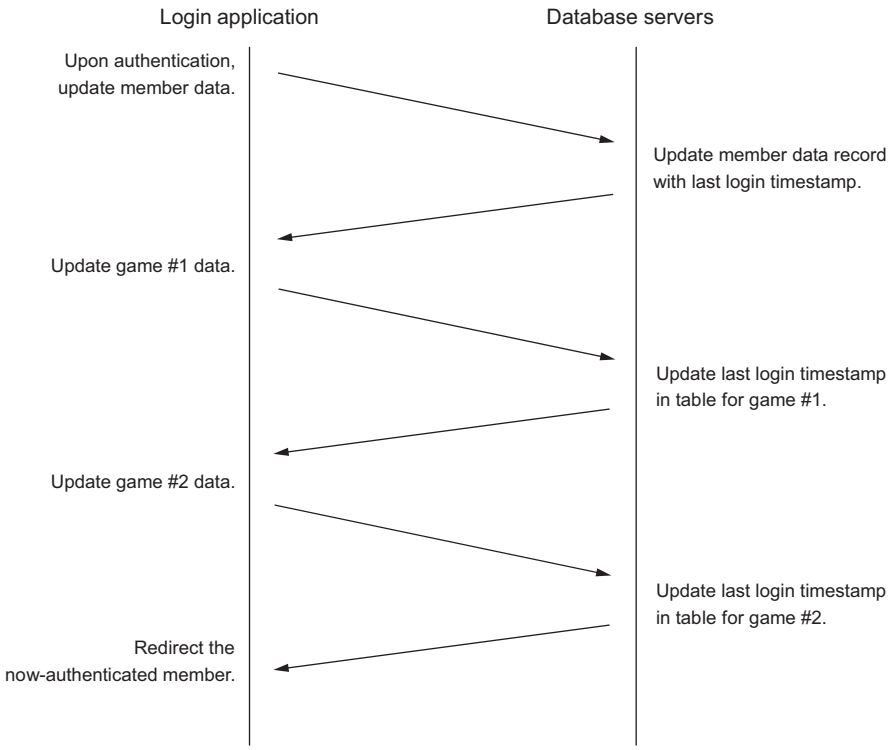

Figure 1.2 Before: once a user has logged in, each database is updated with a timestamp sequentially and dependently. The more tables you add, the longer this takes.

message broker that would then distribute the messages to any number of consumer applications that would do the database writes required. I experimented with several different message brokers, and ultimately I landed on RabbitMQ as my broker of choice.

> **DEFINITION** Message-oriented middleware (MOM) is defined as software or hardware infrastructure that allows for the sending and receiving of messages from distributed systems. RabbitMQ fills this role handily with functionality that provides advanced routing and message distribution, even with wide area network (WAN) tolerances to support reliable, distributed systems that interconnect with other systems easily.

After decoupling the login process from the database updates that were required, I discovered a new level of freedom. Members were able to quickly log in because we were no longer updating the database as part of the authentication process. Instead, a member login message was published containing all of the information needed to update any database, and consumer applications were written that updated each database table independently (figure 1.3). This login message didn't contain authentication information for the member, but instead, only the information needed to maintain the

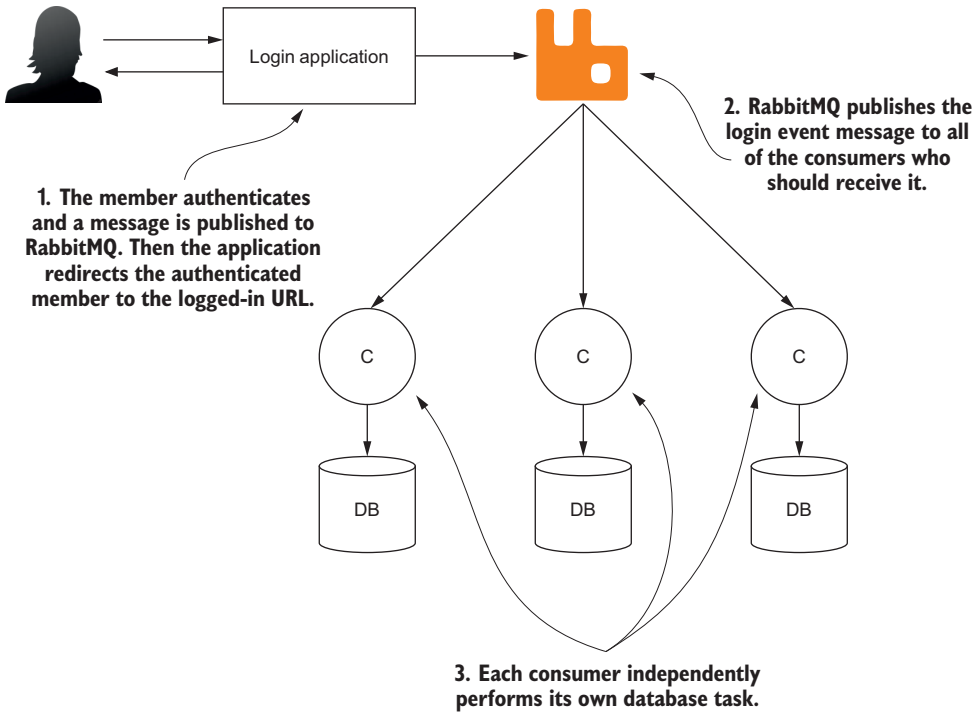

Figure 1.3 After: using RabbitMQ, loosely coupled data is published to each database asynchronously and independently, allowing the login application to proceed without waiting on any database writes.

member's last login status in our various databases and applications. This allowed us to horizontally scale database writes with more control. By controlling the number of consumer applications writing to a specific database server, we were able to throttle database writes for servers that had started to strain under the load created by new site growth while we worked through their own unique scaling issues.

As I detail the advantages of a messaging-based architecture, it's important to note that these advantages could also impact the performance of systems like the login architecture described. Any number of problems may impact publisher performance, from networking issues to RabbitMQ throttling message publishers. When such events happen, your application will see degraded performance. In addition to horizontally scaling consumers, it's wise to plan for horizontal scaling of message brokers to allow for better message throughput and publisher performance.

1.3.1 Decoupling your application

The use of messaging-oriented middleware can provide tremendous advantages for organizations looking to create flexible application architectures that are data centric. By moving to a loosely coupled design using RabbitMQ, application architectures are no

longer bound to database write performance and can easily add new applications to act upon the data without touching any of the core applications. Consider figure 1.4, demonstrating the design of a tightly coupled application communicating with a database.

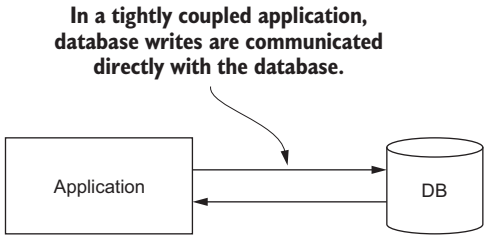

Figure 1.4 When communicating with a database, a tightly coupled application must wait for the database server to respond before it can continue processing.

1.3.2 Decoupling database writes

In a tightly coupled architecture, the application must wait for the database server to respond before it can finish a transaction. This design has the potential to create performance bottlenecks in both synchronous and asynchronous applications. Should the database server slow down due to poor tuning or hardware issues, the application will slow. Should the database stop responding or crash, the application will potentially crash as well.

By decoupling the database from the application, a loosely coupled architecture is created. In this architecture, RabbitMQ, as messaging-oriented middleware, acts as an intermediary for the data prior to some action being taken with it in the database. A consumer application picks up the data from the RabbitMQ server, performing the database action (figure 1.5).

In this model, should a database need to be taken offline for maintenance, or should the write workload become too heavy, you can throttle the consumer application or stop it. Until the consumer is able to receive the message, the data will persist

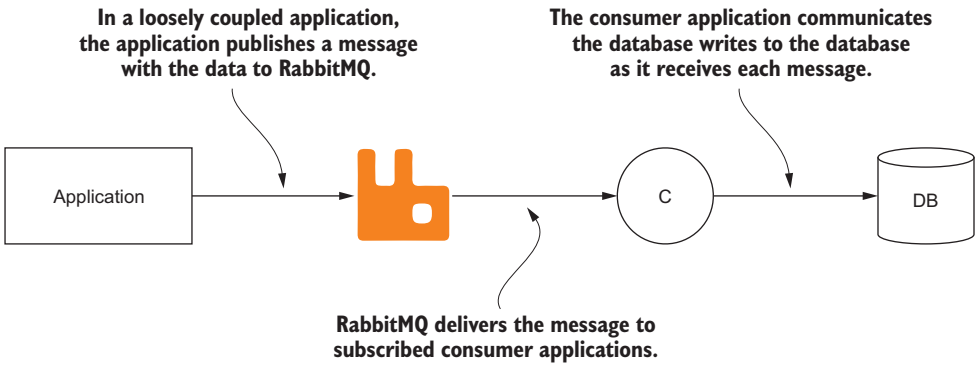

Figure 1.5 A loosely coupled application allows the application that would have saved the data directly in the database to publish the data to RabbitMQ, allowing for the asynchronous processing of data.

in the queue. The ability to pause or throttle consumer application behavior is just one advantage of using this type of architecture.

1.3.3 Seamlessly adding new functionality

Loosely coupled architectures leveraging RabbitMQ allow data to be repurposed as well. The data that originally was only going to be written to a database can also be used for other purposes. RabbitMQ will handle all of the duplication of message content and can route it to multiple consumers for multiple purposes (figure 1.6).

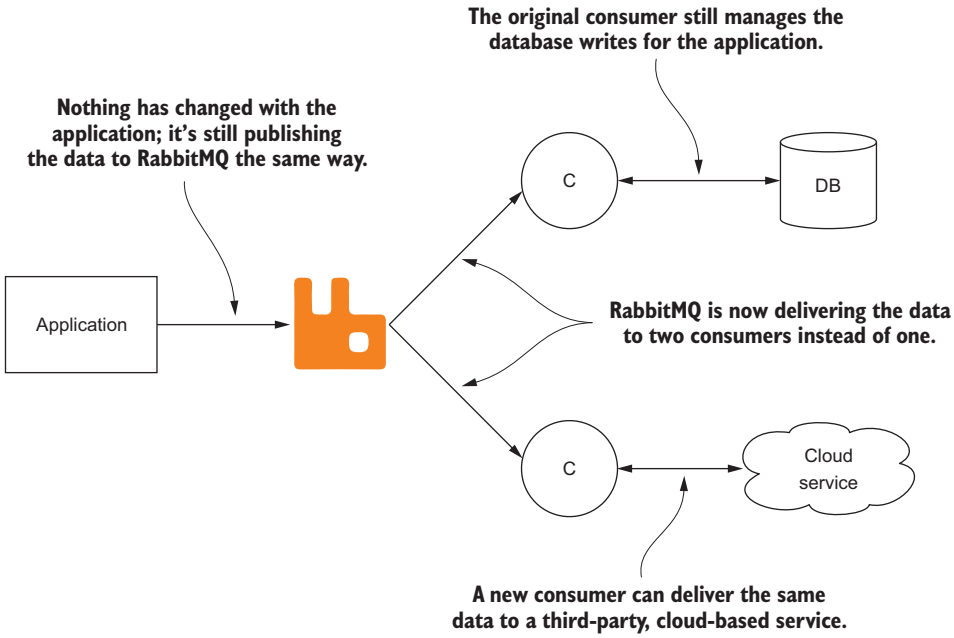

Figure 1.6 By using RabbitMQ, the publishing application doesn't need to be changed in order to deliver the same data to both a new cloud-based service and the original database.

1.3.4 Replication of data and events

Expanding upon this model, RabbitMQ provides built-in tools for cross–data center distribution of data, allowing for federated delivery and synchronization of applications. Federation allows RabbitMQ to push messages to remote RabbitMQ instances, accounting for WAN tolerances and network splits. Using the RabbitMQ federation plugin, it's easy to add a RabbitMQ server or cluster in a second data center. This is illustrated in figure 1.7, where the data from the original application can now be processed in two different locations over the internet.

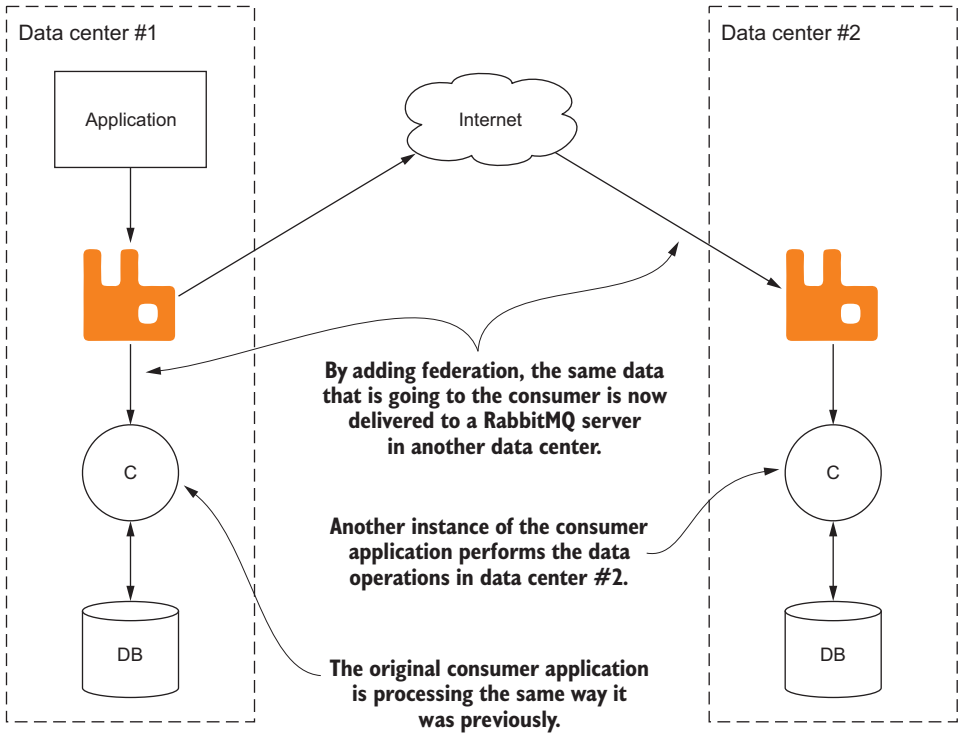

Figure 1.7 By leveraging RabbitMQ's federation plugin, messages can be duplicated to perform the same work in multiple data centers.

1.3.5 Multi-master federation of data and events

Expanding upon this concept by adding the same front-end application to a second data center and setting the RabbitMQ servers to bidirectionally federate data, you can have highly available applications in different physical locations. Messages from the application in either data center are sent to consumers in both data centers, allowing for redundancy in data storage and processing (figure 1.8). This approach to application architecture can allow applications to scale horizontally, also providing geographic proximity for users and a cost-effective way to distribute your application infrastructure.

> **NOTE** As with any architecture decision, using messaging-oriented middleware introduces a degree of operational complexity. Because a message broker becomes a center point in your application design, a new single point of failure is introduced. There are strategies, which we'll cover in this book, to create highly available solutions to minimize this risk. In addition, adding a message broker creates a new application to manage. Configuration, server resources, and monitoring must be taken into account when weighing the tradeoffs of introducing a message broker to your architecture. I'll teach you how to account for these and other concerns as you proceed through the book.

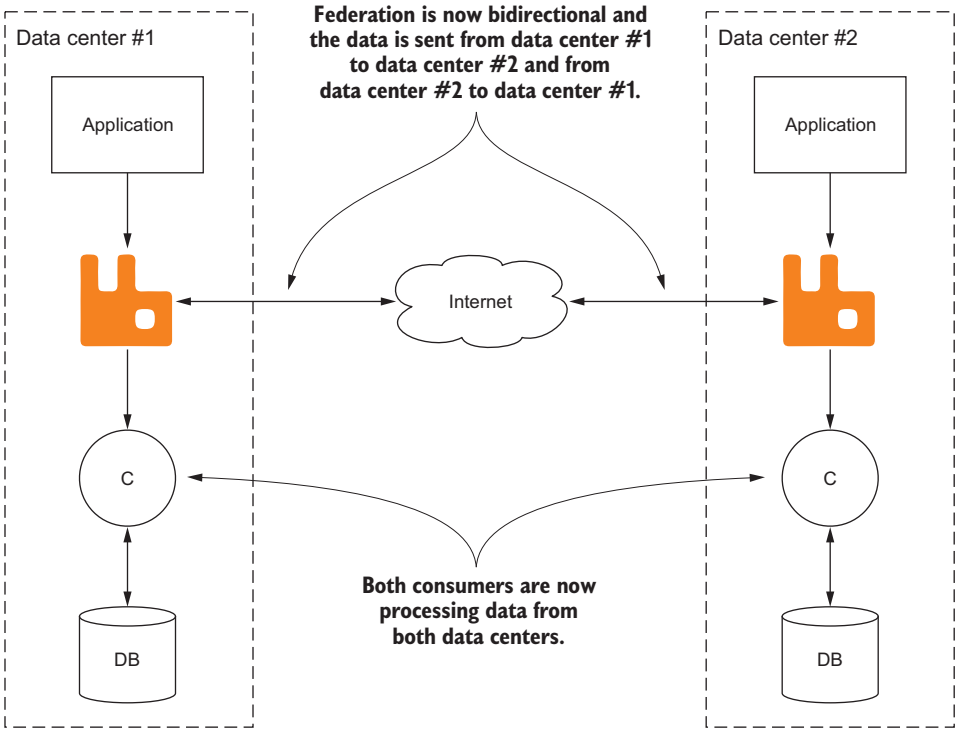

Figure 1.8 Bidirectional federation of data allows for the same data events to be received and processed in both data centers.

1.3.6 The Advanced Message Queuing model

Many of RabbitMQ's strengths, including its flexibility, come from the AMQP specification. Unlike protocols like HTTP and SMTP, the AMQP specification defines not only a network protocol but also server-side services and behaviors. I'll refer to this information as the Advanced Message Queuing (AMQ) model. The AMQ model logically defines three abstract components in broker software that define the routing behavior of messages:

- *Exchange*—The component of the message broker that routes messages to queues
- *Queue*—A data structure on disk or in memory that stores messages
- *Binding*—A rule that tells the exchange which queue the messages should be stored in

The flexibility of RabbitMQ comes from the dynamic nature of how messages can be routed through exchanges to queues. The bindings between exchanges and queues, and the message routing dynamics they create, are a foundational component of implementing a messaging-based architecture. Creating the right structure using these basic tools in RabbitMQ allows your applications to scale and easily change with the underlying business needs.

The first piece of information that RabbitMQ needs in order to route messages to their proper destination is an exchange to route them through.

Exchanges

Exchanges are one of three components defined by the AMQ model. An exchange receives messages sent into RabbitMQ and determines where to send them. Exchanges define the routing behaviors that are applied to messages, usually by examining data attributes passed along with the message or that are contained within the message's properties.

RabbitMQ has multiple exchange types, each with different routing behaviors. In addition, it offers a plugin-based architecture for custom exchanges. Figure 1.9 shows a logical view of a publisher sending a message to RabbitMQ, routing a message through an exchange.

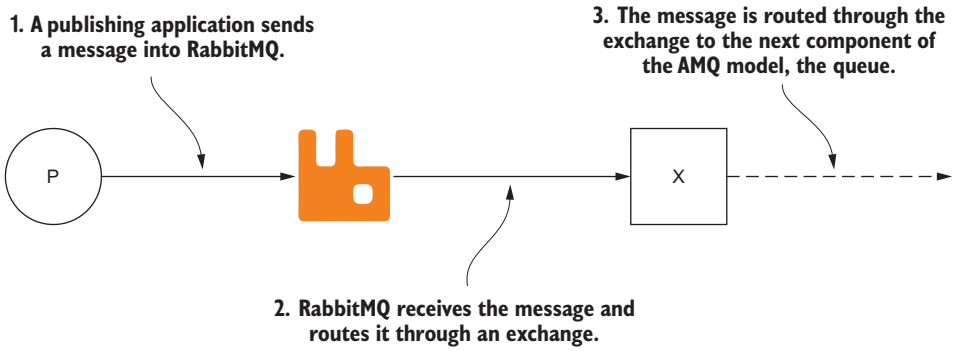

Figure 1.9 When a publisher sends a message into RabbitMQ, it first goes to an exchange.

Queues

A queue is responsible for storing received messages and may contain configuration information that defines what it's able to do with a message. A queue may hold messages in RAM only, or it may persist them to disk prior to delivering them in first-in, first-out (FIFO) order.

Bindings

To define a relationship between queues and exchanges, the AMQ model defines a *binding*. In RabbitMQ, bindings or *binding keys*, tell an exchange which queues to deliver messages to. For some exchange types, the binding will also instruct the exchange to filter which messages it can deliver to a queue.

When publishing a message to an exchange, applications use a *routing*-key attribute. This may be a queue name or it may be a string that semantically describes the message. When a message is evaluated by an exchange to determine the appropriate queues it should be routed to, the message's routing key is evaluated against the binding

key (figure 1.10). In other words, the binding key is the glue that binds a queue to an exchange, and the routing key is the criteria that's evaluated against it.

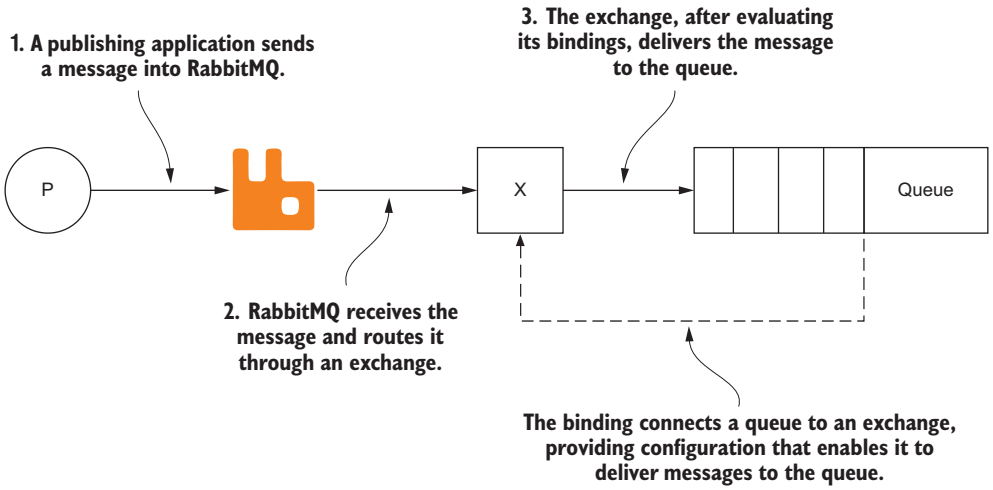

Figure 1.10 A queue is bound to an exchange, providing the information the exchange needs to route a message to it.

In the most simple of scenarios, the routing key may be the queue name, though this varies with each exchange type. In RabbitMQ, each exchange type is likely to treat routing keys in a different way, with some exchanges invoking simple equality checks and others using more complex pattern extractions from the routing key. There's even an exchange type that ignores the routing key outright in favor of other information in the message properties.

In addition to binding queues to exchanges, as defined in the AMQ model, RabbitMQ extends the AMQP specification to allow exchanges to bind to other exchanges. This feature creates a great deal of flexibility in creating different routing patterns for messages. In addition to the various routing patterns available when you use exchanges, you'll learn more about exchange-to-exchange bindings in chapter 6.

1.4 Summary

RabbitMQ, as messaging-oriented middleware, is an exciting technology that enables operational flexibility that's difficult to achieve without the loosely coupled application architecture it enables. By diving deep into RabbitMQ's AMQP foundation and behaviors, this book should prove to be a valuable reference, providing insight into how your applications can leverage its robust and powerful features. In particular, you'll soon learn how to publish messages and use the dynamic routing features in RabbitMQ to selectively sip from the fire hose of data your application can send, data that once may have been deeply buried in tightly coupled code and processes in your environment.

Whether you're an application developer or a high-level application architect, it's advantageous to have a deep level of knowledge about how your applications can benefit from RabbitMQ's diverse functionality. Thus far, you've learned the most foundational concepts that comprise the AMQ model. I'll expand on these concepts in the remainder of part 1 of this book: You'll learn about AMQP and how it defines the core of RabbitMQ's behavior.

Because this book will be hands-on, with the goal of imparting the knowledge required to use RabbitMQ in the most demanding of environments, you'll start working with code in the next chapter. By learning "how to speak Rabbit," you'll be leveraging the fundamentals of AMQP, writing code to send and receive messages with RabbitMQ. To speak Rabbit, you'll be using a Python-based library called *rabbitpy,* a library that was written specifically for the code examples in this book; I'll introduce it to you in the next chapter. Even if you're an experienced developer who has written applications that communicate with RabbitMQ, you should at least browse through the next chapter to understand what's happening at the protocol level when you're using RabbitMQ via the AMQP protocol.

SOFTWARE DEVELOPMENT

RabbitMQ IN DEPTH

Gavin M. Roy • Technical Editor James Titcumb

At the heart of most modern distributed applications is a queue that buffers, prioritizes, and routes message traffic. RabbitMQ is a high-performance message broker based on the Advanced Message Queueing Protocol. It's battle tested, ultrafast, and powerful enough to handle anything you can throw at it. It requires a few simple setup steps, and you can instantly start using it to manage low-level service communication, application integration, and distributed system message routing.

RabbitMQ in Depth is a practical guide to building and maintaining message-based applications. This book provides detailed coverage of RabbitMQ with an emphasis on why it works the way it does. You'll find examples and detailed explanations based in real-world systems ranging from simple networked services to complex distributed designs. You'll also find the insights you need to make core architectural choices and develop procedures for effective operational management.

What's Inside

- AMQP, the Advanced Message Queueing Protocol
- Communicating via MQTT, Stomp, and HTTP
- Valuable troubleshooting techniques
- Database integration

Written for programmers with a basic understanding of messaging-oriented systems.

Gavin M. Roy is an active, open source evangelist and advocate who has been working with internet and enterprise technologies since the mid-90s. Technical editor **James Titcumb** is a freelance developer, trainer, speaker, and active contributor to open source projects.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/rabbitmq-in-depth

"An excellent resource for beginners and experts alike ... shows how to integrate RabbitMQ into a successful enterprise application."
—Ian Dallas, Hewlett-Packard

"The most comprehensive source for everything RabbitMQ. From terms to code to patterns, it's all here!"
—Andrew Meredith
Quantum Metric

"A cheat sheet for getting started and troubleshooting the migration process to RabbitMQ."
—Nadia Noori
La Salle University Barcelona

"Filled with pragmatic advice and pearls of wisdom."
—Miloš Milivojević, Mozzart Bet

ISBN-13: 978-1-61729-100-5
ISBN-10: 1-61729-100-5
55999
9 781617 291005

MANNING $59.99 / Can $79.99 [INCLUDING eBOOK]