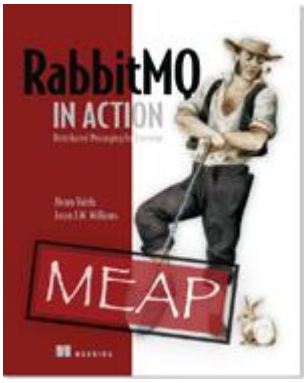
MANNING PUBLICATIONS



RabbitMQ in Action
By Alvaro Videla and Jason J.W. Williams

Message queuing simply is connecting your applications together with messages that are routed between them by a message broker like RabbitMQ. This green paper, based on chapter 1 of RabbitMQ in Action, introduces the history of messaging and RabbitMQ.

Pulling RabbitMQ Out of the Hat

We live in a world where real-time information is constantly around us. Where the applications we write need easy ways to be routed to multiple receivers reliably and quickly. More importantly, we need ways to change who gets the information our apps create without constantly rewriting them. Too often, our application's information becomes siloed, unable to be accessed by new programs that need it without rewriting (and probably breaking) the original producers. You might be saying to yourself, "Sure, but how can message queuing or RabbitMQ help me fix that?"

Well, my friend, let me start by asking you if this scenario sounds familiar: You've just finished implementing a great authentication module for your company's killer web app. It's beautiful. On every page hit, your code efficiently coordinates with the authentication server to make sure your users can only access what they should. You're feeling pretty smug because every page hit on your company's world-class avocado distribution website activates your code. That's about the time your boss walks in and tells you the company needs a way to log every successful and failed permission attempt so that it can be data mined. After lightly protesting that that's the job of the authentication server, your boss not so gently informs you that there's no way to access that data. The authentication server logs it in a proprietary format; hence, this is now your problem Mr. Über Developer. Mulling the situation causes a four aspirin headache as you realize you're going to have to modify your authentication module and probably break every page in the process. After all, that wonderful code of yours touches EVERY access to the site.

Let's stop for a moment though. Let's punch the Easy Button and time-warp back to the beginning of the development of that great auth module. Let's assume you leveraged message queuing heavily in its design from day one. With RabbitMQ in place, you brilliantly leveraged message queuing to decouple your module from the authentication server. With every page request, your authentication module is designed to place an authorization request message into RabbitMQ. The authentication server then listens on a RabbitMQ queue that receives that request message. Once the request is approved, the auth server puts a reply message back into RabbitMQ where it is routed to the queue that your amazing module is listening on.

In this world, your boss's request doesn't even faze you. You realize you don't need to touch your module or even write a new one. All you need to do is write a small app that connects to RabbitMQ and subscribes to the authorization requests your auth module is already publishing. No code changes. Nothing you already wrote knows anything has changed. It's so simple a smile almost breaks out on your face. That's the power of messaging to make your day job easier.

Message queuing simply is connecting your applications together with messages that are routed between them by a message broker like RabbitMQ. It's like putting in a post office just for your applications. The reality is that this approach isn't just a solution to the real-time problems of the financial industry either; it's a solution to the problems we all face as developers every day. Your authors here don't come from a financial services background. We had no idea what *enterprise messaging* was when we needed to scale. No sir, we were simply devs like you with an itch that needed scratching. An itch to deal with real-time volumes of information and route it to multiple consumers quickly. We needed to do it all without blocking the producers of that information...and without them needing to know who the final consumers might be. RabbitMQ helped us to solve those common problems easily and in a standards-based way that ensures any app of ours can talk to any other app be it Python, PHP, or even Scala.

Before we're done with this article, you'll know the short history of messaging and RabbitMQ. Without further ado, let's take a look at where all this messaging fun started.

Living in other people's dungeons...

The world of message queuing that we live in didn't start out the dank and cramped one it is today, with most folks subservient to lock-in overlords. In fact, it started with a ray of light in an otherwise byzantine software landscape. It was 1983 when a 26-year old engineer from Mumbai had a radical idea: why wasn't there a common software *bus*? A communication system that would do the heavy lifting of communicating information from one interested application to another. Coming from an education in hardware design at MIT, Vivek Ranadivé envisioned a common bus like the one on a motherboard; only this one would be a software bus that applications could plug into (*Driving the Information Bus*). Thus, in 1983 Teknekron was born with Ranadivé at its helm. A freshly-minted Harvard MBA in his hand and this powerful idea in his head, Vivek started plowing a path that would help developers everywhere.

Having the idea was one thing; however, finding a killer application for it was something completely different. It was at Goldman Sachs in 1985 that Ranadivé found his first customer and the problem his software bus was born to solve: financial trading. A trader's stall at that time was packed to the brim with different terminals for each type of information the trader needed to do his job. What Teknekron saw was an opportunity to replace all those terminals and their siloed applications. In their place would be Ranadivé's *software bus*. What would remain would be a single workstation whose display programs could now plug into the Teknekron *software bus* as consumers and allow the trader to *subscribe* to the information the trader wanted to see. Publish-subscribe (PubSub) was born, as was the world's first modern message queuing software: Teknekron's *The Information Bus* (TIB).

It didn't take very long for this model of data transfer to find many more killer uses. After all, an application publishing data and an application consuming it no longer had to directly connect to each other. Heck, they didn't even have to know the other existed. What Teknekron's TIB allowed application developers to do was to establish a set of rules for describing the message content. As long as the messages were published according to those rules, any consuming application could subscribe to a copy of the messages tagged with topics it was interested in.

Producers and consumers of information could now be completely decoupled and flexibly mixed on the fly. Both sides of the PubSub model (producer/consumer) were completely interchangeable without breaking the opposite side. The only thing that needed to remain stable was the TIB software and the rules for tagging and routing the information. Since the financial trading industry is full of information with a constantly changing set of interested folks, TIB spread like wildfire in that sector. It was also noticed by telecommunications and especially news organizations, who also had information that needed timely delivery to a dynamically changing set of interested consumers. That's why mega news outfit Reuters purchased Teknekron in 1994.

Meanwhile, this burgeoning new segment of enterprise software didn't go unnoticed by Big Blue. After all, many of IBM's biggest customers were in the financial services industry. Also, Teknekron's TIB software was frequently run on IBM hardware and operating systems...all without the boys in White Plains getting a cut. Thus, in the late 80s, IBM began research into developing their own message-queuing software, leveraging their extensive experience in information delivery from developing DB2. Development began in 1990 at IBM's Hursley Park Laboratories near Winchester, United Kingdom. What emerged 3 years later in 1993 was the IBM MQseries™ family of message queuing-server software. In the 17 years since, MQseries has evolved into WebSphere MQ(tm) and is

today the dominant commercial message queuing platform. During that time, Ranadivé's TIB hardly disappeared into the bowels of Reuters. Instead, it has remained the other major player in *enterprise messaging*, thriving through a renaming to Rendezvous™ and Teknekron's re-emergence as an independent company in the form of TIBCO in 1997. The very same year, Microsoft's first crack at the messaging market emerged: Microsoft Message Queue™, aka MSMQ.

Through all of this evolution, message-queuing (MQ) software largely remained the exclusive domain of large-budgeted organizations with a need for reliable, decoupled, real-time message delivery. Why didn't the MQ find a larger audience?

How did it survive the information boom that was the late 90s Internet bubble without finding explosive adoption? After all, everyone today from Twitter to Salesforce.com is scrambling to create internal solutions to the PubSub problems that The Information Bus solved 25 years ago. Two words: vendor lock-in. The commercial MQ vendors wanted to help applications interoperate, not create standard interfaces that would allow different MQ products to interoperate or, heaven forbid, allow applications to change MQ platforms. Vendor lock-in has kept prices and margins high and commercial MQ software out of reach of the startups and Web 2.0 companies that are exploding today.

As it turned out, smaller tech companies weren't the only ones unhappy about the high-priced walled gardens of MQ vendors. The financial services companies that formed the bread and butter of the MQ industry weren't thrilled either.

Inevitably, the size of financial companies meant that there were MQ products in place from multiple vendors servicing different internal applications. If an application subscribing to information on a TIBCO, MQ suddenly needed to consume messages from an IBM MQ, it couldn't easily be done. They used different APIs, different wire protocols and definitely couldn't be federated together into a single bus. From this problem the Java Message Service (JMS) was born in 2001. JMS attempted to solve the lock-in and interoperability problem by providing a common Java API that hides the actual interface to the individual vendor MQ products. Technically, a Java application only needs to be written to the JMS API, and the appropriate MQ drivers selected.

JMS takes care of the rest...supposedly. The problem is you're trying to glue a single standard interface over multiple very diverse ones. It's like gluing together different types of cloth: eventually the seams come apart and the reality breaks through. Applications could become more brittle with JMS, not less. A new standards-based approach to messaging was needed.

AMQP to the rescue

In 2004, JPMorgan Chase required a better solution to the problem and started the development of the Advanced Message Queuing Protocol (AMQP) with iMatix Corporation. AMQP from the get-go was designed to be an open standard that would solve the vast majority of message queuing needs and topologies. By virtue of being an open standard, anyone can implement it and anyone who codes to the standard can interoperate with MQ servers from any AMQP vendor.

In many ways, AMQP promises to liberate us from the dungeons of vendors and fulfill the original 1985 vision of Ranadivé: dynamically connecting information in real time from any publisher to any interested consumer over a software bus.

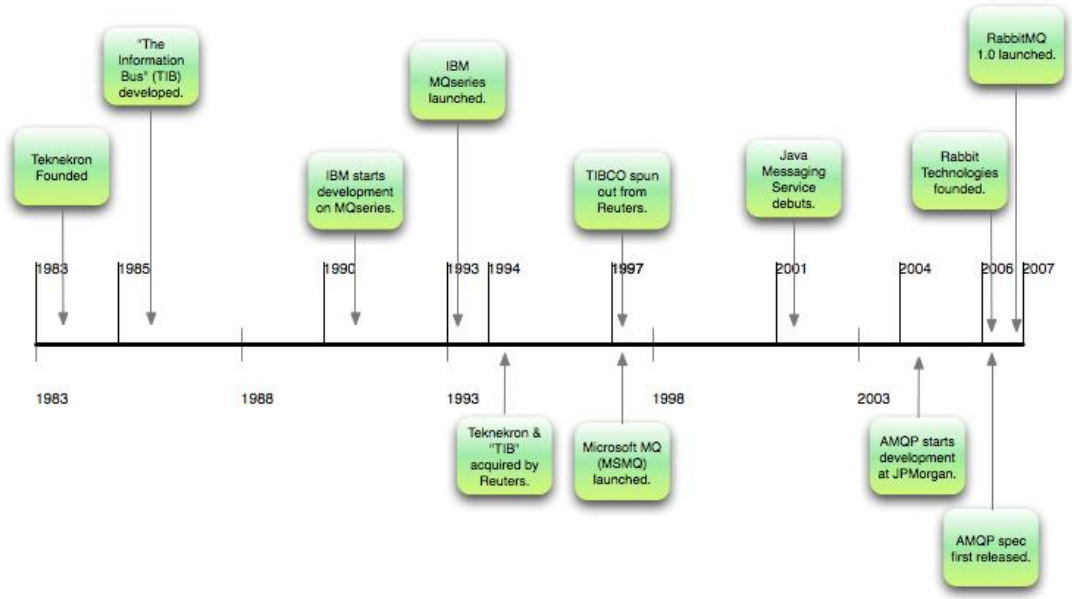


Figure 1 short timeline of message queuing

A Brief History of RabbitMQ

In the early 2000s, a young entrepreneur out of the London financial sector co-founded a company for caching Java objects: Metalogic. For Alexis Richardson, the theory was simple enough: use Java objects for distributed computing and cache them in transit for performance. The reality however, was far different.

Varying versions of the Java Virtual Machine as well as differing libraries on the client and server could make the objects unusable when they arrived. There were just too many environment variables in the real world for Metalogic's approach to be widely successful. What did come out of Metalogic, however, was Alexis's meeting with Matthias Radestock.

Matthias was working for LShift, where Alexis was subleasing office space while at Metalogic. LShift at the time was heavily involved in language modeling and distributed computing contracts for a major software vendor. The background in these areas triggered Matthias' interest in Erlang, the programming language that Ericsson had originally developed for their telephone switching gear. What grabbed Matthias' attention was that Erlang excelled at distributed programming and robust failure recovery but, unfortunately, at the time, it wasn't open source. In the meantime, Metalogic had closed operations and LShift was in the process of winding down their primary distributed computing contract. However, Alexis had learned two very valuable lessons from his experience at Metalogic: 1) what works in a distributed computing environment and 2) what companies want for those environments.

Alexis knew he wanted to start a new company to solve the problems of communicating in a distributed environment. He also knew the next company he started would be open source and build on the model just proving successful by JBoss and MySQL. Looking back at the where the Metalogic solution had run into problems, Alexis started to see more and more that messaging was the right answer to distributed computing. More importantly, in the tech world circa 2004, there was a huge hole for open source messaging. No one was providing a messaging solution except for the big commercial vendors, and while *enterprise* open source was flourishing with databases (MySQL) and application servers (JBoss), no one was touching the missing component: messaging. Interestingly, it was in 2004 that AMQP was just starting to be developed at JPMorgan Chase. Through his background in the financial industry, Alexis had been introduced to the principal driver of AMQP at JPMorgan, John O'Hara (future founder of the AMQP Working Group). It was through O'Hara that Alexis became acquainted with AMQP, and started lining up the building blocks for what would become RabbitMQ.

Around 2005, Alexis cofounded CohesiveFT. He and his cofounders in the U.S. started the company to provide an application stack and tools for what has today become cloud computing. That a key part of that stack would be distributed messaging seemed obvious to Alexis, who (still in the same office as LShift) started talking about AMQP to Matthias. What was clear about AMQP to Matthias when he saw it was that he'd just found the application he'd be looking to write in Erlang. However, before any of this could get started, Alexis and Matthias focused on three questions that they knew would be critical to an open-source version of AMQP's success if it were written in Erlang:

- Would large financial institutions care if their messaging broker was written in Erlang?
- Was Erlang really a good choice for writing an AMQP server?
- If it was written in Erlang, would that slow down adoption in the open-source community?

Issue 1 was quickly dispatched by a financial company who confirmed they didn't care what it was written in if it helped reduce their integration costs. The second question was answered by Francesco Cesarini at Erlang Solutions: from his analysis of AMQP, the specification implied an architecture present in every telephone switch. In other words, you couldn't pick a better implementation language than Erlang for building an AMQP broker. The final question was put to rest by an entirely different messaging server: ejabberd. Extensible Messaging and Presence Protocol (XMPP) by 2005 had become a respected standard for open instant messaging, and one of the foremost implementations was the Erlang-based ejabberd server package by Alexey Shchepin. ejabberd was widely in use by many different organizations and its implementation in Erlang didn't seem to be slowing anyone down.

With the three major questions answered, Alexis and Matthias convinced CohesiveFT and LShift to jointly back the project. The first thing they did was to contract Matthew Sackman (now a core Rabbit developer) to write a prototype in Erlang to test latency. What they quickly discovered was that using the distributed computing libraries built into Erlang produced incredible latency that was comparable to using raw sockets. There was, of course, also the question of what to call this thing...which everyone agreed was Rabbit. After all, rabbits are fast and they multiply like crazy, making it a great name for distributed software. Not the least of the reasons, however: Rabbit is easy to remember. Thus, in 2006, Rabbit Technologies was born: a joint-venture between CohesiveFT and LShift that would hold the intellectual property for what we know today as RabbitMQ.

The timing couldn't have been more perfect because around the same time, the first public draft of the AMQP specification had become available. As a new specification, AMQP was rapidly changing. This was an area where Erlang proved critical. By using Erlang, RabbitMQ could be developed quickly and keep pace with what was a moving target: the AMQP standard. Amazingly, version 1.0 of RabbitMQ was written in only two and a half months by core developer Tony Garnock-Jones. From the beginning, RabbitMQ has implemented a key feature of AMQP that differentiates it from TIBCO and IBM: provisioning resources like queues and exchanges can be done from within the protocol itself. With the commercial vendors, provisioning is done by specialized staff at specialized administrative consoles. With RabbitMQ those tasks can be done by the application itself, making it the perfect communication bus for anyone building a distributed application, particularly, one that leverages cloud-based resources and rapid deployment.

That brings us to today, where RabbitMQ is used by everyone from small Silicon Valley startups to some of the largest names in the Internet. That's perhaps the best thing about RabbitMQ...and the thing that surprised its founders: its largest block of users are tech firms, not financial companies. RabbitMQ fulfills the vision of Ranadivé for the rest of us with smaller budgets and the same real problems.

That's what drew the authors to RabbitMQ. We didn't know that what we were looking for was message queuing software. All we knew was that we had real problems to solve integrating applications and serving higher and higher transaction loads. RabbitMQ provides a powerful toolkit for solving those problems and brings to the masses the rich history of messaging and, finally, a pluggable information bus for everyone that needs one.

Today, RabbitMQ isn't the only game in town for open messaging. You have options like ActiveMQ, ZeroMQ, and Apache Qpid—all providing different open-source approaches to message queuing. The question is why do we think you should pick RabbitMQ?

- Except for Qpid, RabbitMQ is the only one implementing the AMQP open standard.
- Clustering is ridiculously simple on RabbitMQ because of Erlang.

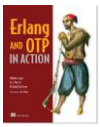
- Your mileage may vary, but we've found RabbitMQ to be far more reliable and crash resistant than its competitors.

Perhaps the most important reason is that RabbitMQ is ridiculously simple to install and use. Whether you need a simple one-node setup for your workstation, or a seven-server cluster to power your web infrastructure, RabbitMQ can be up and running in about 30 minutes.

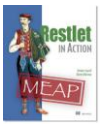
Summary

Now you can see why we love RabbitMQ so much. Despite being the progeny of technology from the financial industry, it's dead simple to set up. You get complex routing and reliability features pioneered by folks like TIBCO and IBM but in a package that's easier to manage and use. And, the best part—it's open source! We've shown how far messaging has come in the past 30 years, from a simple software bus linking together financial traders, to message routing monsters that are the beating heart of everything from financial exchanges to the manufacturing lines at semiconductor fabs.

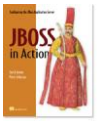
Here are some other Manning titles you might be interested in:



Erlang and OTP in Action
Martin Logan, Eric Merritt, and Richard Carlsson



Restlet in Action
Jerome Louvel and Thierry Boileau



JBoss in Action
Configuring the JBoss Application Server
Javid Jamae and Peter Johnson

Last updated: March 18, 2011