Reactive Design Patterns

Roland Kuhn
with Brian Hanafee and Jamie Allen
Foreword by Jonas Bonér

MANNING

Sample Chapter

Reactive Design Patterns

by Roland Kuhn

with Brian Hanafee and Jamie Allen

Chapter 2

brief contents

PART 1 INTRODUCTION 1

1 ■ Why Reactive? 3
2 ■ A walk-through of the Reactive Manifesto 12
3 ■ Tools of the trade 39

PART 2 THE PHILOSOPHY IN A NUTSHELL 65

4 ■ Message passing 67
5 ■ Location transparency 81
6 ■ Divide and conquer 91
7 ■ Principled failure handling 100
8 ■ Delimited consistency 105
9 ■ Nondeterminism by need 113
10 ■ Message flow 120

PART 3 PATTERNS 125

11 ■ Testing reactive applications 127
12 ■ Fault tolerance and recovery patterns 162
13 ■ Replication patterns 184
14 ■ Resource-management patterns 220
15 ■ Message flow patterns 255
16 ■ Flow control patterns 294
17 ■ State management and persistence patterns 311

2 A walk-through of the Reactive Manifesto

This chapter introduces the manifesto in detail: where the original text is as short as possible and rather dense, we unfold it and discuss it in great depth. For more background information on the theory behind the manifesto, please refer to part 2 of the book.

2.1 Reacting to users

So far, this book has used the word *user* informally and mostly in the sense of humans who interact with a computer. You interact only with your web browser in order to read and write emails, but many computers are needed in the background to perform these tasks. Each of these computers offers a certain set of services, and the consumer or user of these services will in most cases be another computer that is acting on behalf of a human, either directly or indirectly.

The first layer of services is provided by the front-end server and consumed by the web browser. The browser makes requests and expects responses—predominantly using HTTP, but also via WebSockets. The resources that are requested can pertain to emails, contacts, chats, searching, and many more (plus the definition of the styles and layout of the website). One such request might be related to the images of people you correspond with: when you hover over an email address, a pop-up window appears that contains details about that person, including a photograph or an avatar image. In order to render that image, the web browser makes a request to the front-end server. Figure 2.1 shows how this might be implemented using a traditional servlets approach.

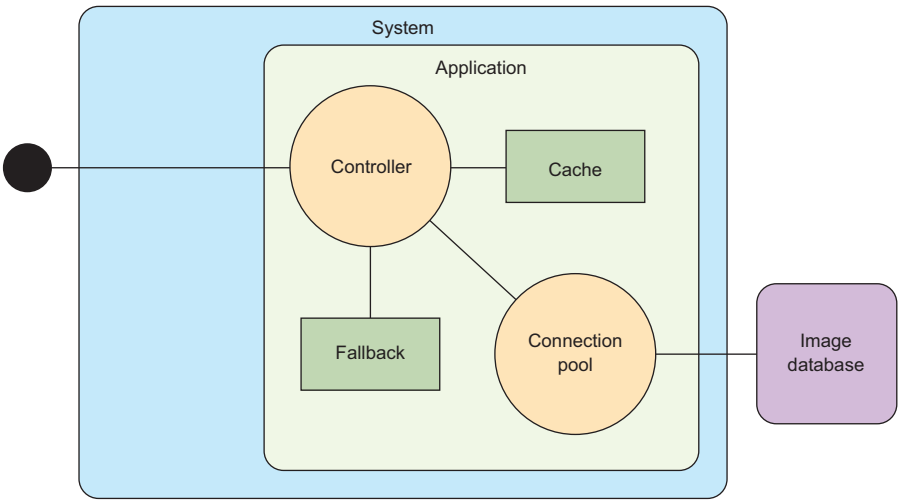

Figure 2.1 The front-end server for images first checks an in-memory cache, then attempts to retrieve the image from storage, and finally returns a fallback image if neither is successful.

The user action of hovering the mouse pointer over an email address sets in motion a flurry of requests via the web browser, the front-end server, and the internal image service down to the storage system, followed by their respective responses traveling in the opposite direction until the image is properly rendered on the screen. Along this chain are multiple relationships from user to service, and all of them need to meet the basic challenges outlined in the introduction; most important is the requirement to respond quickly to each request.

When designing the overall implementation of a feature like the image service, you need to think about services and their users' requirements not only on the outside but also on the inside. This is the first part of what it means to build reactive applications. Once the system has been decomposed in this way, you need to turn your focus to making these services as responsive as necessary to satisfy their users at all levels.

To understand why Reactive systems are better than the traditional alternatives, it is useful to examine a traditional implementation of an image service. Even though it has a cache, a connection pool, and even a fallback image for when things go wrong, it can fail badly when the system is stressed. Understanding how and why it fails requires looking beyond the single-thread illusion. Once you understand the failures, you will see that even within the confines of a traditional framework, you can improve the image service with a simplified version of the Managed Queue pattern that is covered in chapter 16.

2.1.1 Understanding the traditional approach

We will start with a naive implementation to retrieve an image from a database. The application has a controller that first checks a cache to see whether the image has

been retrieved recently. If the controller finds the image in the cache, it returns the image right away. If not, it tries to retrieve the image from the database. If it finds the image there, the image is added to the cache and also returned to the original requester. If the image is not found, a static fallback image is returned, to avoid presenting the user with an error. This pattern should be familiar to you. This simplistic controller might contain code like the following.

Listing 2.1 Excerpt from a simple controller for an image service

```
public interface Images {
  Image get(String Key);
  void add(String key, Image image);
}

public Images cache;          <-- Assumed thread-safe
public Images database;       <-- Wraps a database connection pool

Image result = cache.get(key);
if (result != null) {
  return result;              <-- Image is found in the cache
} else {
  result = database.get(key);
  if (result != null) {
    cache.add(key, result);   <-- Image is found in the database, added to the cache, and returned to the client
    return result;
  } else {
    return fallback;          <-- Image is not retrieved from the database
  }
}
```

At the next level of detail, the application may be built on a framework that has some ability to handle concurrency, such as Java servlets. When a new request is received, the application framework assigns it to a request thread. That thread is then responsible for carrying the request through to a response. The more request threads are configured, the more simultaneous requests the system is expected to handle.

On a cache hit, the request thread can provide a response immediately. On a cache miss, the internal implementation of `Images` needs to obtain a connection from the pool. The database query itself may be performed on the request thread, or the connection pool may use a separate thread pool. Either way, the request thread is obliged to wait for the database query to complete or time out before it can fulfill the request.

When you are tuning the performance of a system such as this, one of the key parameters is the ratio of request threads to connection-pool entries. There is not much point in making the connection pool larger than the request-thread pool. If it is the same size and all the request threads are waiting on database queries, the system may find itself temporarily with little to do other than wait for the database to respond. That would be unfortunate if the next several requests could have been served from the cache; instead of being handled immediately, they will have to wait for

an unrelated database query to complete so that a request thread will become available. On the other hand, setting the connection pool too small will make it a bottleneck; this risks the system being limited by request threads stuck waiting for a connection.

The best answer for a given load is somewhere between the extremes. The next section looks at finding a balance.

2.1.2 Analyzing latency with a shared resource

The simplistic implementation can be analyzed first by examining one extreme consisting of an infinite number of request threads sharing a fixed number of database connections. Assume each database query takes a consistent time W to complete, and for now ignore the cache. We will revisit the effect of the cache in section 2.3.1, when we introduce Amdahl's Law. You want to know how many database connections L will be used for a given load, which is represented as λ. A formula called *Little's Law* gives the answer:

$$L = \lambda \times W$$

Little's Law is valid for the long-term averages of the three quantities independent of the actual timing with which requests arrive or the order in which they are processed. If the database takes on average 30 ms to respond, and the system is receiving 500 requests per second, you can apply Little's Law:

$$L = 500 \text{ requests/second} \times 0.03 \text{ seconds/request}$$

$$L = 15$$

The average number of connections being used will be 15, so you will need at least that many connections to keep up with the load.

If there are requests waiting to be serviced, they must have some place to wait. Typically, they wait in a queue data structure somewhere. As each request is completed, the next request is taken from the queue for processing. Referring to figure 2.2, you may notice that there is no explicit queue. If this were coded using traditional synchronous Java servlets, the queue would consist of an internal collection of request threads waiting for their turn with the database connection. On average, there would be 15 such threads waiting. That is not good, because, whereas a queue is a lightweight data structure, the request threads in the queue are relatively expensive resources. Worse, 15 is just the average: the peaks are much higher. In reality, the thread pool will not be infinite. If there are too many requests, they will spill back into the TCP buffer and eventually back to the browser, resulting in unhelpful errors rather than the desired fallback image.

The first thing you might do is increase the number of entries in the database connection pool. As long as the database can continue to handle the resulting load, this will help the average case. The important thing to note is that you are still working with

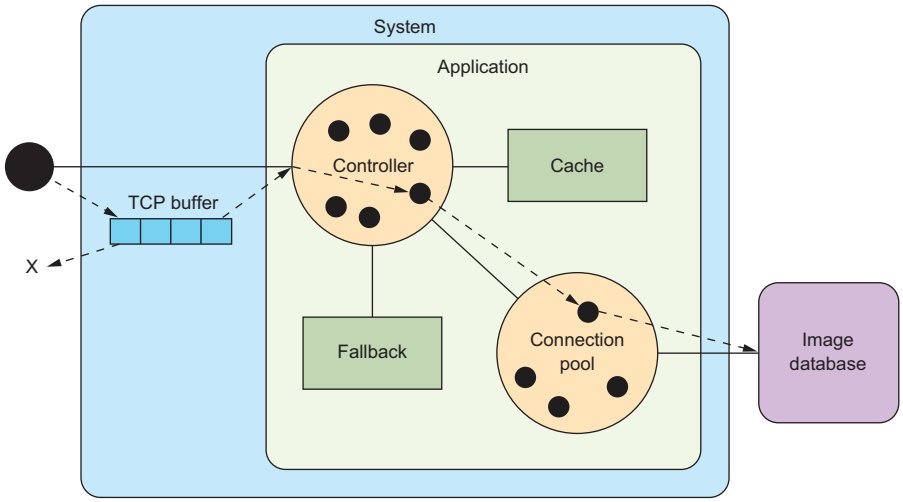

Figure 2.2 Using standard listener threads and a connection pool results in the listeners acting as queue entries, with overflow into the system TCP buffers.

average times. Real-world events can lead to failure modes that are far worse. For example, if the database stops responding at all for several minutes, 500 requests per second will overwhelm an otherwise sufficient thread pool. You need to protect the system.

2.1.3 Limiting maximum latency with a queue

The initial implementation blocked and waited for a database connection to become available; it returned `null` only if the requested image was not found in the database. A simple change will add some protection: if a database connection is not available, return `null` right away. This will free the request thread to return the fallback image rather than stalling and consuming a large amount of resources.

This approach couples two separate decisions into one: the number of database queries that can be accepted simultaneously is equal to the size of the connection pool. That may not be the result you want: it means the system will either return right away if no connection is available or return in 30 ms if one is available. Suppose you are willing to wait a bit longer in exchange for a much better rate of success. At this point, you can introduce an explicit queue, as shown in figure 2.3. Now, instead of returning right away if no connection is available, new requests are added to the queue. They are turned away only if the queue itself is full.

The addition provides much better control over system behavior. For example, a queue with a maximum length of only 3 entries will respond in no more than a total of 120 ms, including 90 ms progressing through the queue and another 30 ms for the database query. The size of the queue provides an upper bound that you can control. Depending on the rate of requests, the average response may be lower, perhaps less

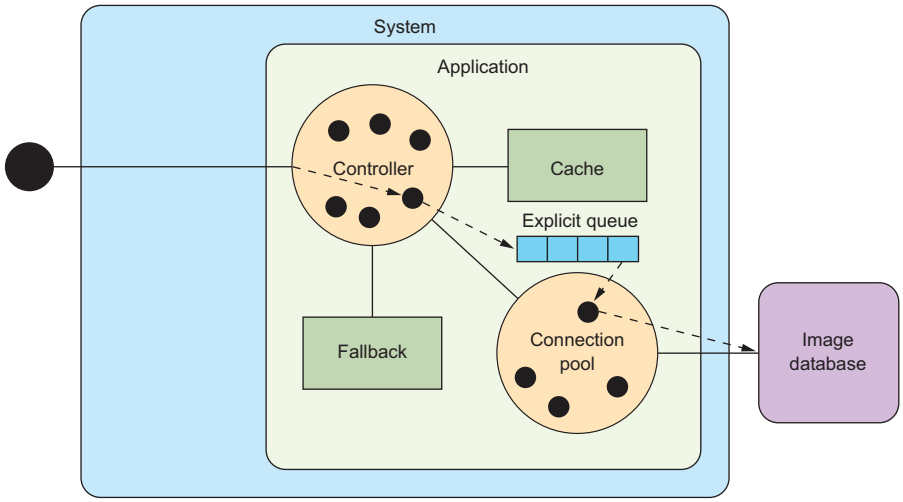

Figure 2.3 Adding an explicit queue to manage access to the database connection pool allows you to manage the maximum system latency separately from the listener thread pool size and the connection pool size.

than 100 ms. If the cache that was ignored in the analysis is now considered, the average drops still further. With a 50% cache-hit rate, the image server could offer an average response time of less than 50 ms.

Given what you know about how that 50 ms average is achieved, you also would know not to set a timeout less than 120 ms. If that time was not acceptable, the simpler solution would be to use a smaller queue. A developer who knows only that the average is less than 50 ms might assume it is a Gaussian distribution and be tempted to set a timeout value at perhaps 80 or 100 ms. Indeed, the assumptions that went into this analysis are vulnerable to the same error, because the assumption that the database provides a *consistent* 30 ms response time would be questionable in a real-world implementation. Real databases have caches of their own.

Setting a timeout has the effect of choosing a boundary at which the system will be considered to have failed. Either the system succeeded or it failed. When viewed from that perspective, the average response time is less important than the maximum response time. Because systems typically respond more slowly when under heavy load, a timeout based on the average will result in a higher percentage of failures under load and will also waste resources when they are needed most. Choosing timeouts will be revisited in section 2.4 and again in chapter 11. For now, the important realization is that the average response time often has little bearing on choosing the maximum limits.

2.2 Exploiting parallelism

The simplest case of a user–service relationship is invoking a method or function:

```
val result = f(42)
```

The user provides the argument "42" and hands over control of the CPU to the function `f`, which might calculate the 42nd Fibonacci number or the factorial of 42. Whatever the function does, you expect it to return some result value when it is finished. This means that invoking the function is the same as making a request, and the function returning a value is analogous to it replying with a response. What makes this example so simple is that most programming languages include syntax like this, which allows direct usage of the response under the assumption that the function does indeed reply. If that were not to happen, the rest of the program would not be executed, because it could not continue without the response. The underlying execution model is that the evaluation of the function occurs synchronously, on the same thread, and this ties the caller and the callee together so tightly that failures affect both in the same way.

Sequential execution of functions is well supported by all popular programming languages out of the box, as illustrated in figure 2.4 and shown in this example using Java syntax:

```
ReplyA a = computeA();
ReplyB b = computeB();
ReplyC c = computeC();
Result r = aggregate(a, b, c);
```

The sequential model is easy to understand. It was adequate for early computers that had only one processing core, but it necessitates waiting for all the results to be computed by the same resource while other resources remain idle.

2.2.1 Reducing latency via parallelization

In many cases, there is one possibility for latency reduction that immediately presents itself. If, for the completion of a request, several other services must be involved, then the overall result will be obtained more quickly if the other services can perform their

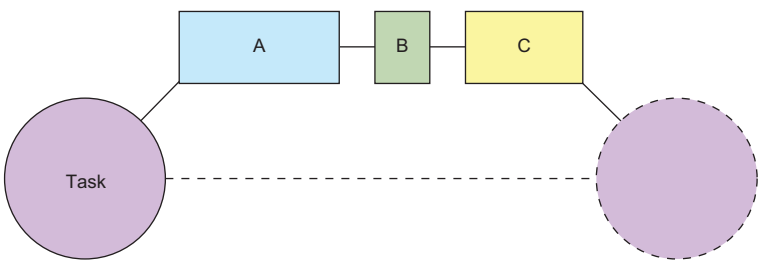

Figure 2.4 A task consisting of three subtasks that are executed sequentially: the total response latency is given by the sum of the three individual latencies.

functions in parallel, as shown in figure 2.5. This requires that no dependency exists such that, for example, task B needs the output of task A as one of its inputs, which frequently is the case. Take as an example the Gmail app in its entirety, which is composed of many different but independent parts. Or the contact information pop-up window for a given email address may contain textual information about that person as well as their image, and these can clearly be obtained in parallel.

When performing subtasks A, B, and C sequentially, as shown in figure 2.4, the overall latency depends on the sum of the three individual latencies. With parallel execution, overall latency equals the latency of whichever of the subtasks takes longest. In the implementation of a real social network, the number of subtasks can easily exceed 100, rendering sequential execution entirely impractical.

Parallel execution usually requires some extra thought and library support. For one thing, the service being called must not return the response directly from the method call that initiated the request, because in that case the caller would be unable to do anything while task A was running, including sending a request to perform task B in the meantime. The way to get around this restriction is to return a *Future* of the result instead of the value itself:

```
Future<ReplyA> a = taskA();
Future<ReplyB> b = taskB();
Future<ReplyC> c = taskC();
Result r = aggregate(a.get(), b.get(), c.get());
```

A Future is a placeholder for a value that may eventually become available; as soon as it does, the value can be accessed via the Future object. If the methods invoking subtasks A, B, and C are changed in this fashion, then the overall task just needs to call them to get back one Future each. Futures are discussed in greater detail in the next chapter.

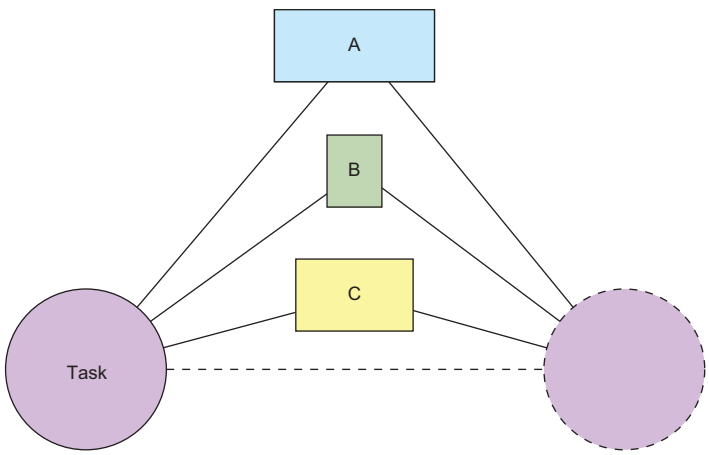

Figure 2.5 A task consisting of three subtasks that are executed in parallel: the total response latency is given by the maximum of the three individual latencies.

The previous code snippet uses a type called `Future` that is defined in the Java standard library (in the package `java.util.concurrent`). The only method it defines for accessing the value is the blocking `get()` method. *Blocking* here means the calling thread is suspended and cannot do anything else until the value has become available. We can picture the use of this kind of Future like so (written from the perspective of the thread handling the overall task):

> When my boss gives me the task to assemble the overview file of a certain client, I will dispatch three runners: one to the client archives to fetch the address, photograph, and contract status; one to the library to fetch all articles the client has written; and one to the post office to collect all new messages for this client. This is a vast improvement over having to perform these tasks myself, but now I need to wait idly at my desk until the runners return, so that I can collate everything they bring into an envelope and hand that back to my boss.
>
> It would be much nicer if I could leave a note telling the runners to place their findings in the envelope and telling the last one to come back to dispatch another runner to hand it to my boss without involving me. That way I could handle many more requests and would not feel useless most of the time.

2.2.2 Improving parallelism with composable Futures

What the developer should do is describe how the values should be composed to form the final result and let the system find the most efficient way to compute the values. This is possible with *composable Futures*, which are part of many programming languages and libraries, including newer versions of Java (`CompletableFuture` is introduced in JDK 8). Using this approach, the architecture turns completely from synchronous and blocking to asynchronous and nonblocking; the underlying machinery needs to become *task-oriented* in order to support this. The result is far more expressive than the relatively primitive precursor, the callback. The previous example transforms into the following, using Scala syntax:[1]

```
val fa: Future[ReplyA] = taskA()
val fb: Future[ReplyB] = taskB()
val fc: Future[ReplyC] = taskC()
val fr: Future[Result] = for (a <- fa; b <- fb; c <- fc)
                         yield aggregate(a, b, c)
```

Initiating a subtask as well as its completion are just events that are raised by one part of the program and reacted to in another part: for example, by registering an action to be taken with the value supplied by a completed Future. In this fashion, the latency of the method call for the overall task does not even include the latencies for subtasks A, B, and C, as shown in figure 2.6. The system is free to handle other requests while

[1] This would also be possible with the Java 8 `CompletionStage` using the `andThen` combinator, but due to the lack of for-comprehensions, the code would grow in size relative to the synchronous version. The Scala expression on the last line transforms to corresponding calls to `flatMap`, which are equivalent to `CompletionStage`'s `andThen`.

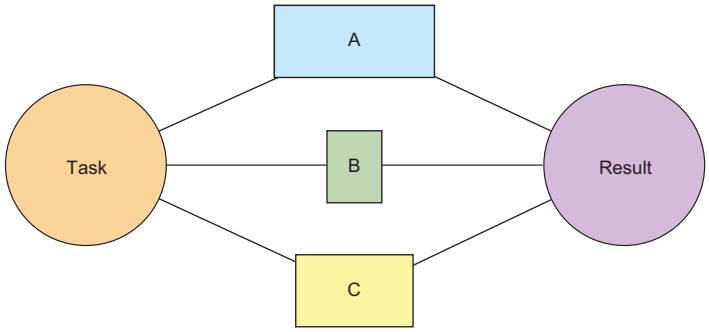

Figure 2.6 A task consisting of three subtasks that are executed as Futures: the total response latency is given by the maximum of the three individual latencies, and the initiating thread does not need to wait for the responses.

those are being processed, eventually reacting to their completion and sending the overall response back to the original user.

An added benefit is that additional events like task timeouts can be added without much hassle, because the entire infrastructure is already there. It is entirely reasonable to perform task A, couple the resulting Future with one that holds a `TimeoutException` after 100 ms, and use the combined result in the processing that follows. Then, either of the two events—completion of A or the timeout—triggers the actions that were attached to the completion of the combined Future.

> **THE NEED FOR ASYNCHRONOUS RESULT COMPOSITION** You may be wondering why this second part—asynchronous result composition—is necessary. Would it not be enough to reduce response latency by exploiting parallel execution? The context of this discussion is achieving bounded latency in a system of nested user–service relationships, where each layer is a user of the service beneath it. Because parallel execution of the subtasks A, B, and C depends on their initiating methods returning Futures instead of strict results, this must also apply to the overall task. That task is very likely part of a service that is consumed by a user at a higher level, and the same reasoning applies on that higher level as well. For this reason, it is imperative that parallel execution be paired with asynchronous and task-oriented result aggregation.

Composable Futures cannot be fully integrated into the image server example discussed earlier using the traditional servlet model. The reason is that the request thread encapsulates all the details necessary to return a response to the browser. There is no mechanism to make that information available to a future result. This is addressed in Servlet 3 with the introduction of `AsyncContext`.

2.2.3 Paying for the serial illusion

Traditionally, ways of modeling interactions between components—like sending to and receiving from the network—are expressed as blocking API calls:

```
final Socket socket = ...
socket.getOutputStream.write(requestMessageBytes);
final int bytesRead = socket.getInputStream().read(responseBuffer);
```

Each of these blocking calls interacts with the network equipment, generating messages and reacting to messages under the hood, but this fact is completely hidden in order to construct a synchronous façade on top of the underlying message-driven system. The thread executing these commands will suspend its execution if not enough space is available in the output buffer (for the first line) or if the response is not immediately available (on the second line). Consequently, this thread cannot do any other work in the meantime: every activity of this type that is ongoing in parallel needs its own thread, even if many of those are doing nothing but waiting for events to occur.

If the number of threads is not much larger than the number of CPU cores in the system, then this does not pose a problem. But given that these threads are mostly idle, you want to run many more of them. Assuming that it takes a few microseconds to prepare the `requestMessageBytes` and a few more microseconds to process the `responseBuffer`, whereas the time for traversing the network and processing the request on the other end is measured in milliseconds, it is clear that each thread spends more than 99% of its time in a waiting state.

In order to fully utilize the processing power of the available CPUs, this means running hundreds if not thousands of threads, even on commodity hardware. At this point, you should note that threads are managed by the operating system kernel for efficiency reasons.[2] Because the kernel can decide to switch out threads on a CPU core at any point in time (for example, when a hardware interrupt happens or the time slice for the current thread is used up), a lot of CPU state must be saved and later restored so that the running application does not notice that something else was using the CPU in the meantime. This is called a *context switch* and costs thousands of cycles[3] every time it occurs. The other drawback of using large numbers of threads is that the scheduler—the part of the kernel that decides which thread to run on which CPU core at any given time—will have a hard time finding out which threads are runnable and which are waiting and then selecting one such that each thread gets its fair share of the CPU.

The takeaway of the previous paragraph is that using synchronous, blocking APIs that hide the underlying message-driven structure wastes CPU resources. If messages were made explicit in the API such that instead of suspending a thread, you would just suspend the computation—freeing up the thread to do something else—then this overhead would be reduced substantially. The following example shows (remote) messaging between Akka Actors from Java 8:

---

[2] Multiplexing several logical user-level threads on a single OS thread is called a *many-to-one model* or *green threads*. Early JVM implementations used this model, but it was abandoned quickly (http://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqh/index.html).

[3] Although CPUs have gotten faster, their larger internal state has negated the advances made in pure execution speed such that a context switch has taken roughly 1 µs without much improvement for two decades.

```
CompletionStage<Response> future =
    ask(actorRef, request, timeout)
      .thenApply(Response.class::cast);
future.thenAccept(response -> <process it>);
```

Sends a message to the actor reference, using a CompletionStage as the destination for the response

Maps the response to its expected type, failing upon mismatch

Registers further processing to be done once a response is received and mapped

Here, the sending of a request returns a handle to the possible future reply—a composable Future, as discussed in chapter 3—to which a callback is attached that runs when the response has been received. Both actions complete immediately, letting the thread do other things after having initiated the exchange.

2.3 The limits of parallel execution

Loose coupling between components—by design as well as at runtime—includes another benefit: more efficient execution. Although hardware used to increase capacity primarily by increasing the computing power of a single sequential execution core, physical limits[4] began impeding progress on this front around 2006. Modern processors now expand capacity by adding ever more cores, instead. In order to benefit from this kind of growth, you must distribute computation even within a single machine. When using a traditional approach with shared state concurrency based on mutual exclusion by way of locks, the cost of coordination between CPU cores becomes very significant.

2.3.1 Amdahl's Law

The example in section 2.1 includes an image cache. The most likely implementation would be a map shared among the request threads running on multiple cores in the same JVM. Coordinating access to a shared resource means executing those portions of the code that depend on the integrity of the map in some synchronized fashion. The map will not work properly if it is being changed at the same time it is being read. Operations on the map need to happen in a *serialized* fashion in some order that is globally agreed on by all parts of the application; this is also called *sequential consistency*. There is an obvious drawback to such an approach: portions that require synchronization cannot be executed in parallel. They run effectively single-threaded. Even if they execute on different threads, only one can be active at any given point in time. The effect this has on the possible reduction in runtime that is achievable by parallelization is captured by Amdahl's Law, shown in figure 2.7.

$$S(n) = \frac{T(1)}{T(N)} = \frac{1}{\alpha + \frac{1-\alpha}{N}} = \frac{N}{1 + \alpha(N-1)}$$

Figure 2.7 Amdahl's Law specifies the maximum increase in speed that can be achieved by adding additional threads.

[4] The finite speed of light as well as power dissipation make further increases in clock frequency impractical.

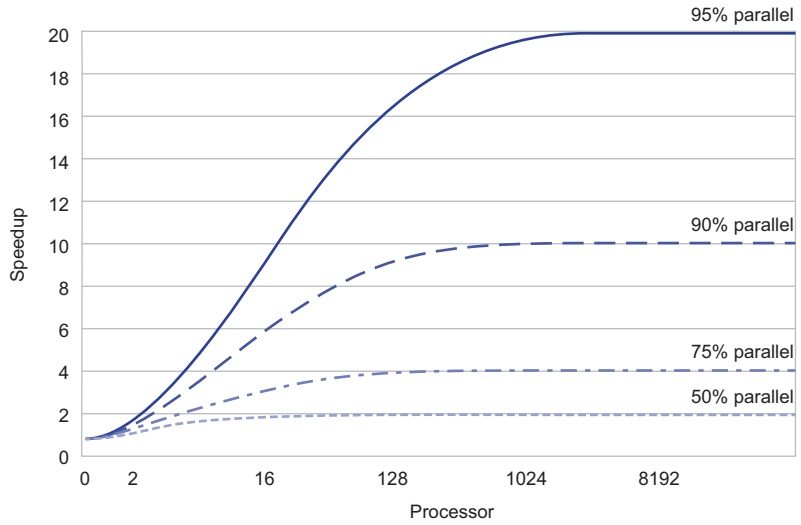

Figure 2.8 The increase in speed of a program using multiple processors in parallel computing is limited by the sequential fraction of the program. For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing will be 20 times, no matter how many processors are used.

Here, N is the number of available threads, α is the fraction of the program that is serialized, and $T(N)$ is the time the algorithm needs when executed with N threads. This formula is plotted in figure 2.8 for different values of α across a range of available threads—they translate into the number of CPU cores on a real system. You will notice that even if only 5% of the program runs inside these synchronized sections, and the other 95% is parallelizable, the maximum achievable gain in execution time is a factor of 20; getting close to that theoretical limit would mean employing the ridiculous number of about 1,000 CPU cores.

2.3.2 Universal Scalability Law

Amdahl's Law also does not take into account the overhead incurred for coordinating and synchronizing the different execution threads. A more realistic formula is provided by the Universal Scalability Law,[5] shown in figure 2.9.

$$S(n) = \frac{N}{1 + \alpha(N-1) + \beta N(N-1)}$$

Figure 2.9 The Universal Scalability Law provides the maximum increase in speed that can be achieved by adding additional threads, with an additional factor to account for coordination.

[5] N. J. Gunther, "A Simple Capacity Model of Massively Parallel Transaction Systems," 2003, www.perfdynamics.com/Papers/njgCMG93.pdf. See also "Neil J. Gunther: Universal Law of Computational Scalability," Wikipedia, https://en.wikipedia.org/wiki/Neil_J._Gunther#Universal_Law_of_Computational_Scalability.

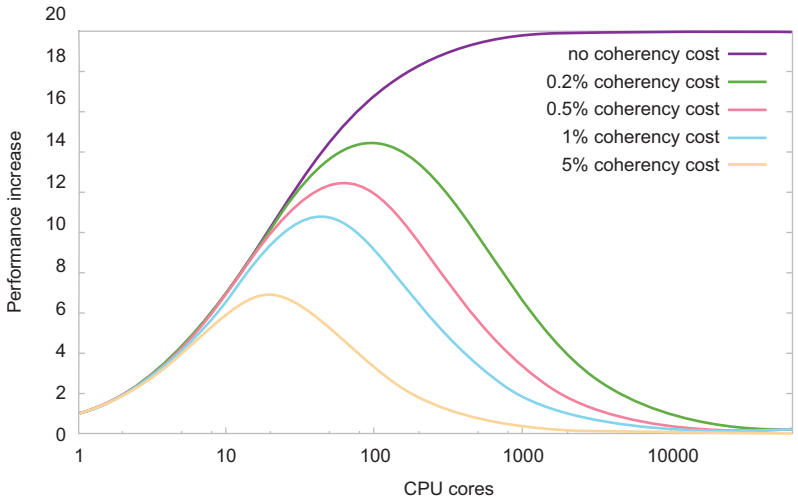

Figure 2.10 At some point, the increase in speed from adding more resources is eaten up by the cost of maintaining coherency within the system. The precise point depends on the parallel program fraction and the time spent on coherency.

The Universal Scalability Law adds another parameter describing the fraction of time spent ensuring that the data throughout the system is consistent. This factor is called the *coherency* of the system, and it combines all the delays associated with coordinating between threads to ensure consistent access to shared data structures. This new term dominates the picture when you have a large number of cores, taking away the throughput benefits and making it unattractive to add more resources beyond a certain point. This is illustrated in figure 2.10 for rather low assumptions on the coherency parameter; distributed systems will spend considerably more than a small percentage of their time on coordination.

The conclusion is that synchronization fundamentally limits the scalability of your application. The more you can do without synchronization, the better you can distribute your computation across CPU cores—or even network nodes. The optimum would be to share nothing—meaning no synchronization would be necessary—in which case scalability would be perfect. In figure 2.9, α and β would be zero, simplifying the entire equation to

$$S(n) = n$$

In plain words, this means that using n times as many computing resources, you achieve n times the performance. If you build your system on fully isolated compartments that are executed independently, then this will be the only theoretical limit, assuming you can split the task into at least n compartments. In practice, you need to exchange requests and responses, which requires some form of synchronization as well, but the cost of that is very low. On commodity hardware, it is possible to exchange several hundred million messages per second between CPU cores.

2.4 Reacting to failure

The previous sections concern designing a service implementation such that every request is met with a response within a given time. This is important because otherwise the user cannot determine whether the request has been received and processed. But even with flawless execution of this design, unexpected things will happen eventually:

- *Software will fail*. There will always be that exception you forgot to handle (or that was not documented by the library you are using); or you may get synchronization only a tiny bit wrong, causing a deadlock to occur; or the condition you formulated for breaking a loop may not cope with a weird edge case. You can always trust the users of your code to figure out ways to eventually find all these failure conditions and more.
- *Hardware will fail*. Everyone who has operated computing hardware knows that power supplies are notoriously unreliable; that hard disks tend to turn into expensive door stops, either during the initial burn-in phase or after a few years; and that dying fans lead to the silent death of all kinds of components by overheating them. In any case, your invaluable production server will, according to Murphy's Law, fail exactly when you most need it.
- *Humans will fail*. When you task maintenance personnel with replacing a failed hard disk in RAID5, a study[6] finds that there is a 10% chance that they will replace the wrong one, leading to the loss of all data. An anecdote from Roland's days as a network administrator is that cleaning personnel unplugged the power of the main server for the workgroup—both redundant cords at the same time—in order to connect the vacuum cleaner. None of these things should happen, but it is human nature that you will have a bad day from time to time.
- *Timeout is failure*. The reason for a timeout may not be related to the internal behavior of the system. For example, network congestion can delay messages between components of your system even when all the components are functioning normally. The source of delay may be some other system that shares the network. From the perspective of handling an individual request, it does not matter whether the cause is permanent or transient. The fact is that the one request has taken too long and therefore has failed.

The question therefore is not *if* a failure occurs but only *when* or *how often*. The user of a service does not care how an internal failure happened or what exactly went wrong, because the only response the user will get is that no normal response is received. Connections may time out or be rejected, or the response may consist of an opaque internal error code. In any case, the user will have to carry on without the response, which for humans probably means using a different service: if you try to book a flight

[6] Aaron B. Brown (IBM Research), "Oops! Coping with Human Error," *ACM Queue* 2, no. 8 (Dec. 6, 2004), http://queue.acm.org/detail.cfm?id=1036497.

and the booking site stops responding, then you will take your business elsewhere and probably not come back anytime soon (or, in a different business, like online banking, users will overwhelm the support hotline).

A high-quality service is one that performs its function very reliably, preferably without any downtime at all. Because failure of computer systems is not an abstract possibility but is in fact certain, the question arises: how can you hope to construct a reliable service? The Reactive Manifesto chooses the term *resilience* instead of *reliability* precisely to capture this apparent contradiction.

What does resilience mean?

Merriam-Webster defines resilience as follows:

- The ability of a substance or object to spring back into shape
- The capacity to recover quickly from difficulties

The key notion here is to aim at fault tolerance instead of fault avoidance, because avoidance will not be fully successful. It is of course good to plan for as many failure scenarios as you can, to tailor programmatic responses such that normal operations can be resumed as quickly as possible—ideally without the user noticing anything. The same must also apply to those failure cases that were not foreseen and explicitly accommodated in the design, knowing that these will happen as well.

But resilience goes one step further than fault tolerance: a resilient system not only withstands a failure but also recovers its original shape and feature set. As an example, consider a satellite that is placed in orbit. In order to reduce the risk of losing the mission, every critical function is implemented at least twice, be it hardware or software. For the case that one component fails, there are procedures that switch to the backup component. Exercising such a fail-over keeps the satellite functioning, but from then on the affected component will not tolerate additional faults because there was only one backup. This means the satellite subsystems are fault tolerant but not resilient.

There is only one generic way to protect your system from failing as a whole when a part fails: *distribute* and *compartmentalize*. The former can informally be translated as "don't put all your eggs in one basket," and the latter adds "protect your baskets from one another." When it comes to handling a failure, it is important to *delegate*, so that the failed compartment itself is not responsible for its own recovery.

Distribution can take several forms. The one you probably think of first involves replicating an important database across several servers such that, in the event of a hardware failure, the data are safe because copies are readily available. If you are really concerned about those data, then you may go as far as placing the replicas in different buildings in order not to lose all of them in the case of fire—or to keep them independently operable when one of them suffers a complete power outage. For the really paranoid, those buildings would need to be supplied by different power grids, better yet in different countries or on separate continents.

2.4.1 Compartmentalization and bulkheading

The further apart the replicas are kept, the smaller the probability of a single fault affecting all of them. This applies to all kinds of failures, whether software, hardware, or human: reusing one computing resource, operations team, set of operational procedures, and so on creates a coupling by which multiple replicas can be affected synchronously or similarly. The idea behind this is to isolate the distributed parts or, to use a metaphor from ship building, to use *bulkheading*.

Figure 2.11 shows the schematic design of a large cargo ship whose hold is separated by bulkheads into many compartments. When the hull is breached for some reason, only those compartments that are directly affected will fill up with water; the others will remain properly sealed, keeping the ship afloat.

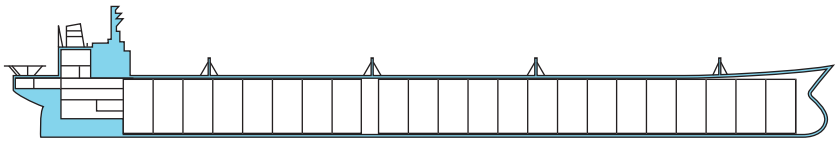

Figure 2.11 The term *bulkheading* comes from ship building and means the vessel is segmented into fully isolated compartments.

One of the first examples of this building principle was the *Titanic*, which featured 15 bulkheads between bow and stern and was therefore considered unsinkable.[7] That particular ship did in fact sink, so what went wrong? In order to not inconvenience passengers (in particular the higher classes) and to save money, the bulkheads extended only a few feet above the water line, and the compartments were not sealable at the top. When five compartments near the bow were breached during the collision with the iceberg, the bow dipped deeper into the water, allowing the water to flow over the top of the bulkheads into more and more compartments until the ship sank.

This example—although certainly one of the most terrible incidents in marine history—perfectly demonstrates that bulkheading can be done wrong in such a way that it becomes useless. If the compartments are not truly isolated from each other, failure can cascade among them to bring down the entire system. One example from distributed computing designs is managing fault tolerance at the level of entire application servers, where one failure can lead to the failure of other servers by overloading or stalling them.

Modern ships employ full compartmentalization where the bulkheads extend from keel to deck and can be sealed on all sides, including the top. This does not make the ships unsinkable, but in order to obtain a catastrophic outcome, the ship needs to be mismanaged severely and run with full speed against a rock.[8] That metaphor translates in full to computer systems.

---

[7] "There is no danger that *Titanic* will sink. The boat is unsinkable and nothing but inconvenience will be suffered by the passengers." —Phillip Franklin, White Star Line vice president, 1912.

[8] See, for example, the *Costa Concordia* disaster: https://en.wikipedia.org/wiki/Costa_Concordia_disaster.

2.4.2 Using circuit breakers

No amount of planning and optimization will guarantee that the services you implement or depend on abide by their latency bounds. We will talk more about the nature of the things that can go wrong when discussing resilience, but even without knowing the source of the failure, there are some useful techniques for dealing with services that violate their bounds.

When users are momentarily overwhelming a service, then its response latency will rise, and eventually it will start failing. Users will receive their responses with more delay, which in turn will increase their own latency until they get close to their own limits. In the image server example in section 2.1.2, you saw how adding an explicit queue protected the client by rejecting requests that would take more than the acceptable response time to service. This is useful when there is a short spike in demand for the service. If the image database were to fail completely for several minutes, the behavior would not be ideal. The queue would fill with a backlog of requests that, after a short time, would be useless to process. A first step would be to cull the old queue entries, but the queue would refill immediately with still more queries that would take too long to process.

In order to stop this effect from propagating across the entire chain of user–service relationships, users need to shield themselves from the overwhelmed service during such time periods. The way to do this is well known in electrical engineering: install a circuit breaker, as shown in figure 2.12.

The idea here is simple: when involving another service, monitor the time it takes for the response to come back. If the time is consistently greater than the allowed threshold this user has factored into its own latency budget for this particular service

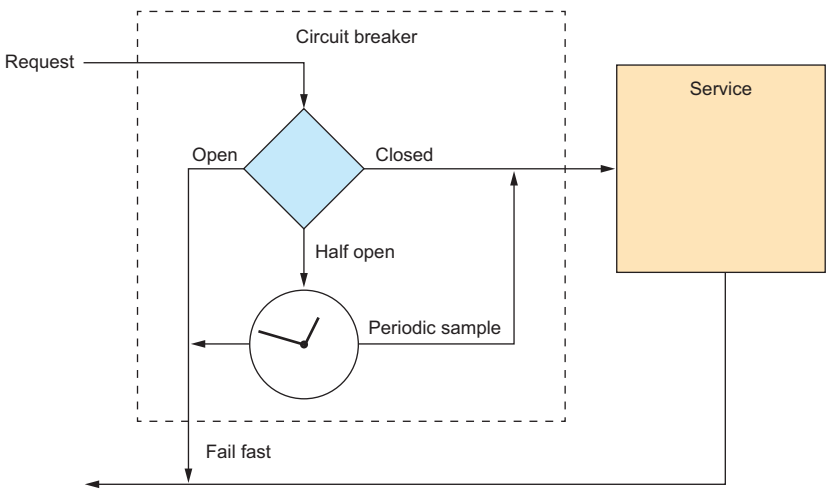

Figure 2.12 A circuit breaker in electrical engineering protects a circuit from being destroyed by a current that is too high. The software equivalent does the same thing for a service that would otherwise be overwhelmed by too many requests.

call, then the circuit breaker trips; from then on, requests will take a different route of processing that either fails fast or gives degraded service, just as in the case of overflowing the bounded queue in front of the service. The same should also happen if the service replies with failures repeatedly, because then it is not worth the effort to send requests.

This not only benefits the user by insulating it from the faulty service but also has the effect of reducing the load on the struggling service, giving it some time to recover and empty its queues. It would also be possible to monitor such occurrences and reinforce the resources for the overwhelmed service in response to the increased load.

When the service has had some time to recuperate, the circuit breaker should snap back into a half-closed state in which some requests are sent in order to test whether the service is back in shape. If not, then the circuit breaker can trip again immediately; otherwise, it closes automatically and resumes normal operations. The Circuit Breaker pattern is discussed in detail in chapter 12.

2.4.3 Supervision

In section 2.2, a simple function call returned a result synchronously:

```
val result = f(42)
```

In the context of a larger program, an invocation of `f` might be wrapped in an exception handler for reasonable error conditions, such as invalid input leading to a divide-by-zero error. Implementation details can result in exceptions that are not related to the input values. For example, a recursive implementation might lead to a stack overflow, or a distributed implementation might lead to networking errors. There is little the user of the service can do in those cases:

```
try {
  f(i)
} catch {
  case ex: java.lang.ArithmeticException => Int.MaxValue
  case ex: java.lang.StackOverflowError => ???
  case ex: java.net.ConnectException => ???
}
```

Reasonable response

Now what?

Responses—including validation errors—are communicated back to the user of a service, whereas failures must be handled by the one who operates the service. The term that describes this relationship in a computer system is *supervision*. The supervisor is responsible for keeping the service alive and running.

Figure 2.13 depicts these two different flows of information. The service internally handles everything it knows how to handle; it performs validation and processes requests, but any exceptions it cannot handle are escalated to the supervisor. While the service is in a broken state, it cannot process incoming requests. Imagine, for example, a service that depends on a working database connection. When the connection breaks, the database driver will throw an exception. If you tried to handle this

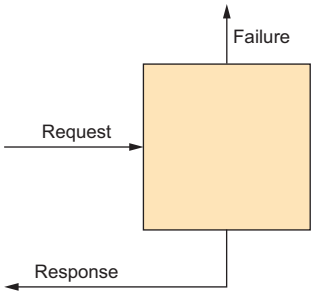

Figure 2.13 Supervision means that normal requests and responses (including negative ones such as validation errors) flow separately from failures: while the former are exchanged between the user and the service, the latter travel from the service to its supervisor.

case directly within the service by attempting to establish a new connection, then that logic would be mixed with all the normal business logic of this service. But, worse, this service would need to think about the big picture as well. How many reconnection attempts make sense? How long should it wait between attempts?

Handing those decisions off to a dedicated supervisor allows the separation of concerns—business logic versus specialized fault handling—and factoring them out into an external entity also enables the implementation of an overarching strategy for several supervised services. The supervisor could, for example, monitor how frequently failures occur on the primary database back-end system and fail over to a secondary database replica when appropriate. In order to do that, the supervisor must have the power to start, stop, and restart the services it supervises: it is responsible for their lifecycle.

The first system that directly supported this concept was Erlang/OTP, implementing the Actor model (discussed in chapter 3). Patterns related to supervision are described in chapter 12.

2.5 Losing strong consistency

One of the most famous theoretical results on distributed systems is Eric Brewer's CAP theorem,[9] which states that any networked shared-data system can have at most two of three desirable properties:

- Consistency (C) equivalent to having a single up-to-date copy of the data
- High availability (A) of that data (for updates)
- Tolerance to network partitions (P)

This means that during a network partition, at least one of consistency and availability must be sacrificed. If modifications continue during a partition, then inconsistencies can occur. The only way to avoid that would be to not accept modifications and thereby be unavailable.

As an example, consider two users editing a shared text document using a service like Google Docs. Hopefully, the document is stored in at least two different locations

[9] S. Gilbert and N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services," *ACM SIGACT News* 33, no. 2 (2002), 51-59, http://dl.acm.org/citation.cfm?id=564601.

in order to survive a hardware failure of one of them, and both users randomly connect to some replica to make their changes. Normally, the changes will propagate between them, and each user will see the other's edits; but if the network link between the replicas breaks down while everything else keeps working, both users will continue editing and see their own changes but not the changes made by the other. If both replace the same word with different improvements, then the result will be that the document is in an inconsistent state that needs to be repaired when the network link starts working again. The alternative would be to detect the network failure and forbid further changes until it is working again—leading to two unhappy users who not only will be unable to make conflicting changes but also will also be prevented from working on completely unrelated parts of the document.

Traditional data stores are relational databases that provide a very high level of consistency guarantees, and customers of database vendors are accustomed to that mode of operation—not least because a lot of effort and research has gone into making databases efficient in spite of having to provide ACID[10] transaction semantics. For this reason, distributed systems have so far concentrated critical components in a way that provides strong consistency.

In the example of two users editing a shared document, a corresponding strongly consistent solution would mean that every change—every keypress—would need to be confirmed by the central server before being displayed locally, because otherwise one user's screen could show a state that was inconsistent with what the other user saw. This obviously does not work, because it would be irritating to have such high latency while typing text—we are used to characters appearing instantly. This solution would also be costly to scale up to millions of users, considering the high-availability setups with log replication and the license fees for the big iron database.

Compelling as this use case may be, Reactive systems present a challenging architecture change: the principles of resilience, scalability, and responsiveness need to be applied to all parts of the system in order to obtain the desired benefits, eliminating the strong transactional guarantees on which traditional systems were built. Eventually, this change will have to occur, though—if not for the benefits outlined in the previous sections, then for physical reasons. The notion of ACID transactions aims at defining a global order of transactions such that no observer can detect inconsistencies. Taking a step back from the abstractions of programming into the physical world, Einstein's theory of relativity has the astonishing property that some events cannot be ordered with respect to each other: if even a ray of light cannot travel from the location of the first event to the location of the second before that event happens, then the observed order of the two events depends on how fast an observer moves relative to those locations.

Although we do not yet need to worry about computers traveling near the speed of light with respect to each other, we do need to worry about the speed of light between

[10] Atomicity, consistency, isolation, durability.

even computers that are stationary. Events that cannot be connected by a ray of light as just described cannot have a causal order between them. Limiting the interactions between systems to proceed, at most, at the speed of light would be a solution to avoid ambiguities, but this is becoming a painful restriction already in today's processor designs: agreeing on the current clock tick on both ends of a silicon chip is one of the limiting factors when trying to increase the clock frequency.

2.5.1 ACID 2.0

Systems with an inherently distributed design are built on a different set of principles. One such set is called BASE:

- Basically available
- Soft state (state needs to be actively maintained instead of persisting by default)
- Eventually consistent

The last point means that modifications to the data need time to travel between distributed replicas, and during this time it is possible for external observers to see data that are inconsistent. The qualification "eventually" means the time window during which inconsistency can be observed after a change is bounded; when the system does not receive modifications any longer and enters a quiescent state, it will eventually become fully consistent again.

In the example of editing a shared document, this means although you see your own changes immediately, you might see the other's changes with some delay; and if conflicting changes are made, then the intermediate states seen by both users may be different. But once the incoming streams of changes end, both views will eventually settle into the same state for both users.

In a note[11] written 12 years after the CAP conjecture, Eric Brewer remarks thus:

> This [see above] expression of CAP served its purpose, which was to open the minds of designers to a wider range of systems and tradeoffs; indeed, in the past decade, a vast range of new systems has emerged, as well as much debate on the relative merits of consistency and availability. The "2 of 3" formulation was always misleading because it tended to oversimplify the tensions among properties. Now such nuances matter. CAP prohibits only a tiny part of the design space: perfect availability and consistency in the presence of partitions, which are rare.

In the argument involving Einstein's theory of relativity, the time window during which events cannot be ordered is very short—the speed of light is rather fast for everyday observations. In the same spirit, the inconsistency observed in eventually consistent systems is also short-lived; the delay between changes being made by one user and being visible to others is on the order of tens or maybe hundreds of milliseconds, which is good enough for collaborative document editing.

[11] Eric Brewer, "CAP Twelve Years Later: How the 'Rules' Have Changed," InfoQ, May 30, 2012, https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed.

BASE has served as an important step in evolving our understanding of which properties are useful and which are unattainable, but as a definitive term it is too imprecise. Another proposal brought forward by Pat Helland at React Conf 2014 is ACID 2.0:

- Associative
- Commutative
- Idempotent
- Distributed

The last point just completes the familiar acronym, but the first three describe underlying mathematical principles that allow operations to be performed in a form that is eventually consistent by definition: if every action is represented such that it can be applied in batches (associative) and in any order (commutative) and such that applying it multiple times is not harmful (idempotent), then the end result does not depend on which replica accepts the change and in which order the updates are disseminated across the network—even resending is fine if reception is not yet acknowledged.

Other authors, such as Peter Bailis and Martin Kleppmann, are pushing the envelope of how far we can extend consistency guarantees without running into the forbidden spot of the CAP theorem: with the help of tracking the causality relationship between different updates, it seems possible to get very close to ACID semantics while minimizing the sacrifice in terms of availability. It will be interesting to see where this field of research will be in 10 years.

2.5.2 Accepting updates

Only during a network partition is it problematic to accept modifications on both disconnected sides, although even for this case solutions are emerging in the form of conflict-free replicated data types (CRDTs). These have the property of merging cleanly when the partition ends, regardless of the modifications that were done on either side.

Google Docs employs a similar technique called *operational transformation*.[12] In the scenario in which replicas of a document get out of sync due to a network partition, local changes are still accepted and stored as operations. When the network connection is back in working condition, the different chains of operations are merged by bringing them into a linearized sequence. This is done by rebasing one chain on top of the other so that instead of operating on the last synchronized state, the one chain is transformed to operate on the state that results from applying the other chain before it. This resolves conflicting changes in a deterministic way, leading to a consistent document for both users after the partition has healed.

Data types with these nice properties come with certain restrictions in terms of which operations they can support. There will naturally be problems that cannot be

[12] David Wang, Alex Mah, and Soren Lassen, "Google Wave Operational Transformation," July 2010, http://mng.bz/Bry5.

stated using them, in which case you have no choice but to concentrate these data in one location only and forgo distribution. But our intuition is that necessity will drive the reduction of these issues by researching alternative models for the respective problem domain, forming a compromise between the need to provide responsive services that are always available and the business-level desire for strong consistency. One example from the real world is automated teller machines (ATMs): bank accounts are the traditional example of strong transactional reasoning, but the mechanical implementation of dispensing cash to account owners has been eventually consistent for a long time.

When you go to an ATM to withdraw cash, you would be annoyed with your bank if the ATM did not work, especially if you needed the money to buy that anniversary present for your spouse. Network problems do occur frequently, and if the ATM rejected customers during such periods, that would lead to lots of unhappy customers—we know that bad stories spread a lot easier than stories that say "It just worked as it was supposed to." The solution is to still offer service to the customer even if certain features like overdraft protection cannot work at the time. You might, for example, get less cash than you wanted while the machine cannot verify that your account has sufficient funds, but you would still get some bills instead of a dire "Out of Service" error. For the bank, this means your account may be overdrawn, but chances are that most people who want to withdraw money have enough to cover the transaction. And if the account has turned into a mini loan, there are established means to fix that: society provides a judicial system to enforce those parts of the contract that the machine could not, and in addition the bank charges fees and earns interest as long as the account holder owes it money.

This example highlights that computer systems do not have to solve all the issues around a business process in all cases, especially when the cost of doing so would be prohibitive. It can also be seen as a system that falls back to an approximate solution until its nominal functionality can be restored.

2.6 The need for Reactive design patterns

Many of the discussed solutions and most of the underlying problems are not new. Decoupling the design of different components of a program has been the goal of computer science research since its inception, and it has been part of the common literature since the famous 1994 *Design Patterns* book.[13] As computers became more and more ubiquitous in our daily lives, programming moved accordingly into the focus of society and changed from an art practiced by academics and later by young "fanatics" in their basements to a widely applied craft. The growth in sheer size of computer systems deployed over the past two decades led to the formalization of designs building on top of the established best practices and widening the scope of what we consider

[13] Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional (1994).

charted territory. In 2003, *Enterprise Integration Patterns*[14] covered message passing between networked components, defining communication and message-handling patterns—for example, implemented by the Apache Camel project. The next step was called *service-oriented architecture* (SOA).

While reading this chapter, you will have recognized elements of earlier stages, such as the focus on message passing and services. The question naturally arises, what does this book add that has not already been described sufficiently elsewhere? Especially interesting is a comparison to the definition of SOA in Arnon Rotem-Gal-Oz's *SOA Patterns* (Manning, 2012):

> DEFINITION: Service-oriented architecture (SOA) is an architectural style for building systems based on interactions of loosely coupled, coarse-grained, and autonomous components called services. Each service exposes processes and behavior through contracts, which are composed of messages at discoverable addresses called endpoints. A service's behavior is governed by policies that are external to the service itself. The contracts and messages are used by external components called service consumers.

This definition focuses on the high-level architecture of an application, which is made explicit by demanding that the service structure be coarse-grained. The reason for this is that SOA approaches the topic from the perspective of business requirements and abstract software design, which without doubt is very useful. But as we have argued, technical reasons push the coarseness of services down to finer levels and demand that abstractions like synchronous blocking network communication be replaced by explicitly modeling the message-driven nature of the underlying system.

2.6.1 Managing complexity

Lifting the level of abstraction has proven to be the most effective measure for increasing the productivity of programmers. Exposing more of the underlying details seems like a step backward on this count, because abstraction is usually meant to hide complications from view. This consideration neglects the fact that there are two kinds of complexity:

- *Essential complexity* is the kind that is inherent in the problem domain.
- *Incidental complexity* is the kind that is introduced solely by the solution.

Coming back to the example of using a traditional database with transactions as the backing store for a shared document editor, the ACID solution tries to hide the essential complexity present in the domain of networked computer systems, introducing incidental complexity by requiring the developer to try to work around the performance and scalability issues that arise.

---

[14] Gregor Hohpe and Bobby Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Addison-Wesley Professional (2003).

A proper solution exposes all the essential complexity of the problem domain, making it accessible to be tackled as is appropriate for the concrete use case, and avoids burdening the user with incidental complexity that results from a mismatch between the chosen abstraction and the underlying mechanics.

This means that as your understanding of the problem domain evolves—for example, recognizing the need for distribution of computation at much finer granularity than before—you need to keep reevaluating the existing abstractions in view of whether they capture the essential complexity and how much incidental complexity they add. The result will be an adaptation of solutions, sometimes representing a shift in which properties you want to abstract over and which you want to expose. Reactive service design is one such shift, which makes some patterns like synchronous, strongly consistent service coupling obsolete. The corresponding loss in level of abstraction is countered by defining new abstractions and patterns for solutions, akin to restacking the building blocks on top of a realigned foundation.

The new foundation is message orientation, and in order to compose large-scale applications on top of it, you need suitable tools to work with. The patterns discussed in the third part of this book are a combination of well-worn, comfortable instruments like the Circuit Breaker pattern as well as emerging patterns learned from wider usage of the Actor model. But a pattern consists of more than a description of a prototypical solution; more important, it is characterized by the problem it tries to solve. The main contribution of this book is therefore to discuss Reactive design patterns in light of the four tenets of the Reactive Manifesto.

2.6.2 Bringing programming models closer to the real world

Our final remark on the consequences of Reactive programming takes up the strands that shone through in several places already. You have seen that the desire to create self-contained pieces of software that deliver service to their users reliably and quickly led to a design that builds on encapsulated, independently executed units of computation. The compartments between the bulkheads form private spaces for services that communicate only using messages in a high-level messaging language.

These design constraints are familiar from the physical world and from our society: humans also collaborate on larger tasks, perform individual tasks autonomously, communicate via high-level language, and so on. This allows us to visualize abstract software concepts using well-known, customary images. We can tackle the architecture of an application by asking, “How would you do it given a group of people?” Software development is an extremely young discipline compared to the organization of labor between humans over the past millennia, and by using the knowledge we have built up, we have an easier time breaking down systems in ways that are compatible with the nature of distributed, autonomous implementation.

Of course, we should stay away from abuses of anthropomorphism: we are slowly eliminating terminology like "master/slave" in recognition that not everybody takes the technical context into account when interpreting them.[15] But even responsible use offers plentiful opportunities for spicing up possibly dull work a little: for example, by calling a component responsible for writing logs to disk a `Scribe`. Implementing that class will have the feel of creating a little robot that will do certain things you tell it to and with which you can play a bit—others call that activity *writing tests* and make a sour face while saying so. With Reactive programming, you can turn this around and realize: it's fun!

2.7 Summary

This chapter laid the foundation for the rest of the book, introducing the tenets of the Reactive Manifesto:

- Responsive
- Resilient
- Elastic
- Message-driven

We have shown how the need to stay responsive in the face of component failure defines resilience, and likewise how the desire to withstand surges in the incoming load elucidates the meaning of scalability. Throughout this discussion, you have seen the common theme of message orientation as an enabler for meeting the other three challenges.

In the next chapter, we will introduce the tools of the trade: event loops, Futures and Promises, Reactive Extensions, and the Actor model. All these make use of the functional programming paradigm, which we will look at first.

[15] Although terminology offers many interesting side notes: for example, a *client* is someone who obeys (from the Latin *cluere*), whereas *server* derives from *slave* (from the Latin *servus*)—so a client–server relationship is somewhat strange when interpreted literally.

An example of naming that can easily prompt out-of-context interpretation is a hypothetical method name like `harvest_dead_children()`. In the interest of reducing nontechnical arguments about code, it is best to avoid such terms.

SOFTWARE DEVELOPMENT

REACTIVE DESIGN PATTERNS

ROLAND KUHN
WITH BRIAN HANAFEE AND JAMIE ALLEN

Modern web applications serve potentially vast numbers of users—and they need to keep working as servers fail and new ones come online, users overwhelm limited resources, and information is distributed globally. A Reactive application adjusts to partial failures and varying loads, remaining responsive in an ever-changing distributed environment. The secret is message-driven architecture—and design patterns to organize it.

Reactive Design Patterns presents the principles, patterns, and best practices of Reactive application design. You'll learn how to keep one slow component from bogging down others with the Circuit Breaker pattern, how to shepherd a many-staged transaction to completion with the Saga pattern, how to divide datasets by Sharding, and much more. You'll even see how to keep your source code readable and the system testable despite many potential interactions and points of failure.

WHAT'S INSIDE

- The definitive guide to the *Reactive Manifesto*
- Patterns for flow control, delimited consistency, fault tolerance, and much more
- Hard-won lessons about what doesn't work
- Architectures that scale under tremendous load

Most examples use Scala, Java, and Akka. Readers should be familiar with distributed systems.

Dr. Roland Kuhn led the Akka team at Lightbend and coauthored the *Reactive Manifesto*. **Brian Hanafee** and **Jamie Allen** are experienced distributed systems architects.

"Does an excellent job explaining Reactive architecture and design, starting with first principles and putting them into a practical context."

—From the Foreword by Jonas Bonér
Creator of Akka

"If the Reactive Manifesto gave us a battle cry, this work gives us the strategic handbook for battle."

—Joel Kotarski, The Rawlings Group

"An engaging tour of distributed computing and the building blocks of responsive, resilient software."

—William Chan, LinkedIn

"This book is so reactive, it belongs on the left-hand side of the periodic table!"

—Andy Hicks, Tanis Systems

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

ISBN-13: 978-1-61729-180-7
ISBN-10: 1-61729-180-3
54999
9 781617 291807

MANNING US $ 49.99 | Can $65.99