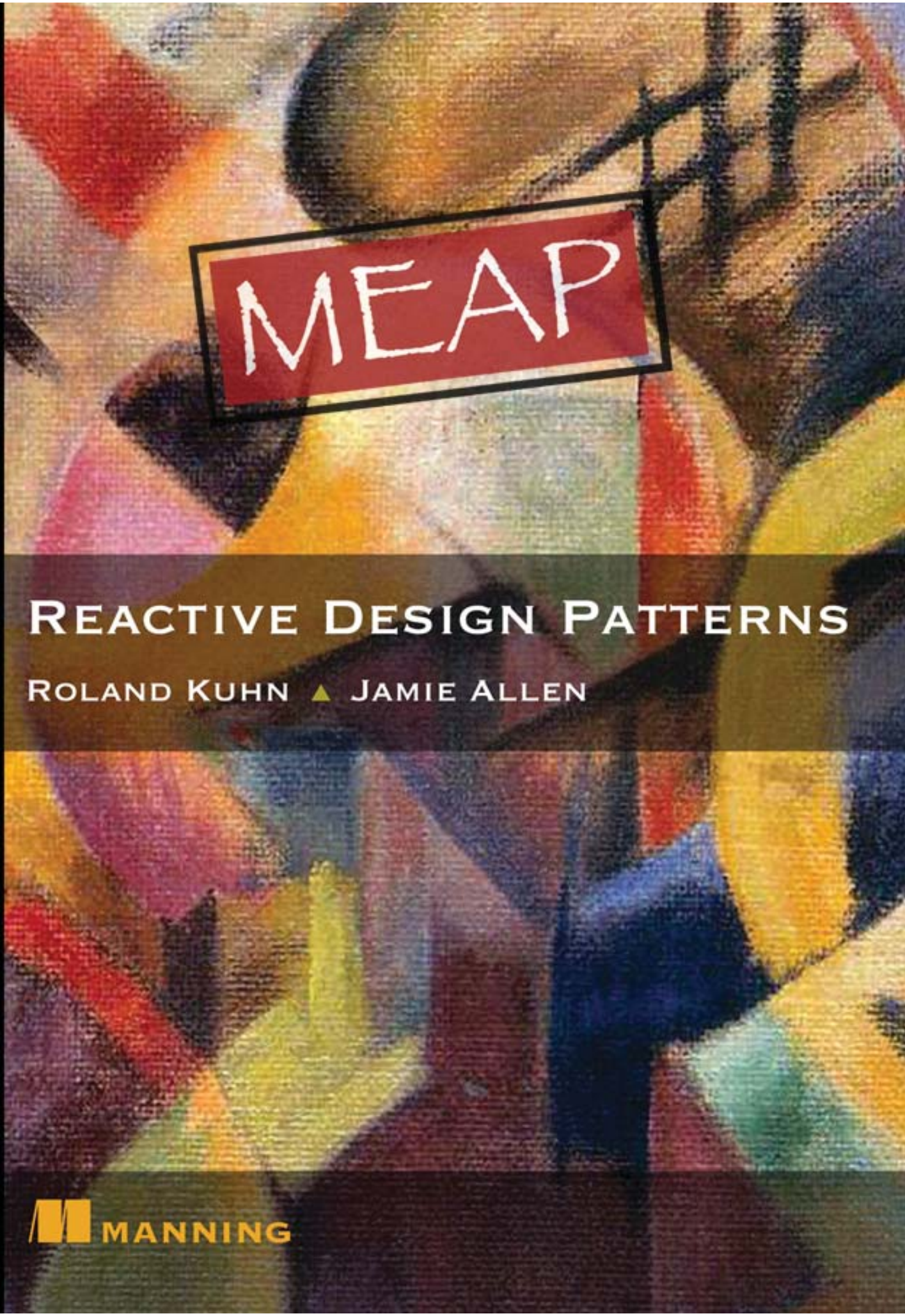
MEAP
REACTIVE DESIGN PATTERNS
ROLAND KUHN ▲ JAMIE ALLEN
MANNING

MEAP Edition
Manning Early Access Program
Reactive Design Patterns
Version 1

For more information on this and other Manning titles go to
www.manning.com

Welcome

Dear Reader,

Thank you for purchasing the MEAP for *Reactive Design Patterns*. We are very excited to have this book ready for the public at a time when Reactive is beginning to blossom in the technical community, and we are greatly looking forward to continuing our work towards its eventual release. This is an intermediate level book for any developer, architect or technical manager who is looking to leverage reactive technologies to deliver the best solutions possible for their users.

We have worked hard to make sure that we not only explain what the primary concerns of reactive applications are, but also what you must do and what tools you must use in order to build a reactive application on your own.

We are initially releasing the first two chapters of the book. Chapter 1 is focused on explaining the reasoning as to why being reactive is so important in application architecture, while chapter 2 delves into the concepts and tools that comprise such applications in detail. Chapter 3 is a deeper dive into the philosophy behind the translation of the four tenets of Reactive Programming into the implementation techniques discussed in the rest of the book.

Looking ahead to the chapters we will be working on next, we will be drilling down into the details of how to leverage each of the concepts in Reactive even further. By the end of Chapter 4, you will have learned on how to test a reactive application to make sure that it is responsive, resilient and scalable. Chapter 5 will discuss fault tolerance and supervision in practice, with Chapter 6 discussing distributed resource management and its impact on consistency.

The final three chapters will focus on best practices and patterns. Chapter 7 will introduce you to patterns focused on flow control in messaging, while Flow Control itself will be addressed in Chapter 8. Chapter 9 will end the book with a discussion of patterns that have proven valuable when writing Actor-based systems.

Both of us will be focused on delivering either a new chapter or an update to an existing one about once a month. As you read the book, we hope you will visit the Author Online forum, where both of us will be reading and responding to your comments and questions. Your feedback will be instrumental in making this best book it possibly can be, and we appreciate any criticism you can provide.

Roland Kuhn
Jamie Allen

brief contents

1 *Why Reactive?*

The purpose of computers is to support us, both in our daily lives and when pushing the boundaries of what is possible. This has been the case since their inception more than a hundred years ago[1]: computers are meant to perform repetitive tasks for us, quickly and without human errors. With this in mind it becomes obvious that the first of the reactive traits—*responsiveness*—is as old as programming itself. When we give a computer a task we want the response back as soon as possible; put another way, the computer must react to its user.

Footnote 1 Charles Babbage's analytical engine was described already 1837, extending the capabilities of his difference engine towards allowing general purpose programs including loops and conditional branches.

For a long time, a single computer was considered fast enough. Complicated calculations like breaking the Enigma chiffre during World War II could take many hours, but the alternative was to not be able to read the enemies' radio transmissions and therefore everyone was very happy with this performance. Today we use computers pervasively in our lives and have become very impatient, expecting responses immediately (or at least within the second). At the same time the tasks we give to computers have become more complex—not in a mathematical sense of pure computation, but in requesting the responses to be distilled from enormous amounts of data. Take for example a web search which requires multiple computers to collaborate on a single task. This principle is also several decades old, we have been using computer networks for over forty years to solve problems that are bigger than one machine alone can handle. But only recently has this architecture been introduced into the design of a single computer in the form of multi-core CPUs, possibly combined in multi-socket servers.

All this taken together means that the distribution of a single program across multiple processor cores—be that within a single machine or across a

network—has become commonplace. The size of such a deployment is defined by how complex the processing task is and by the number of concurrent users to be supported by the system. In former times, the latter number was rather small, in many typical situations one human would use one computer to perform a certain task. Nowadays the internet connects billions of people all across the world and popular applications like social networks, search engines, personal blog services, etc. are used by millions or even billions of users around the clock. This represents a change in both scope and scale of what we expect our computers to do for us, and therefore we need to refine and adapt our common application design and implementation techniques.

In this introduction we started out from the desire to build systems that are *responsive* to their users. Adding the fundamental requirement for distribution is what makes us recognize the need for new (or as so often rediscovered) architecture patterns: in the past we developed band-aids[2] which allowed us to retain the illusion of single-threaded local processing while having it magically executed on multiple cores or network nodes, but the gap between that illusion and reality is becoming prohibitively large. The solution is to make the distributed, concurrent nature of our applications explicit in the programming model, using it to our advantage.

Footnote 2 For example take Java EE services which allow you to transparently call remote services that are wired in automatically, possibly even including distributed database transactions.

This book will teach you how to write systems that stay responsive in the face of variable load, partial outages, program failure and more. We will see that this requires adjustments in the way we think about and design our applications. First—in the rest of this chapter—we take a thorough look at the four tenets of the Reactive Manifesto[3], which was written to define a common vocabulary and to lay out the basic challenges which a modern computer system needs to meet:

Footnote 3 http://reactivemanifesto.org/

- it must react to its users *(responsiveness)*
- it must react to failure and stay available *(resilience)*
- it must react to variable load conditions *(scalability)*
- it must react to events *(event orientation)*

As we will see in the following, the second and third points capture that the system needs to stay responsive in the face of failure or stress, respectively, while the fourth is a consequence of the first three in that the system needs to react to all

kinds of events as they happen, without having a way to know beforehand which to expect next.

In chapters two and three you will get to know several tools of the trade and the philosophy behind their design, which will enable you to effectively use these tools for implementing reactive designs. The different patterns which are emergent from these designs are presented in chapters four to nine using these technologies, therefore it will be helpful to have tried out the different tools you encounter while reading chapter two. This book assumes that you are fluent in Java or Scala, where the latter is used to present code examples throughout the book while Java translations are available for download.

Before we dive right in, we need to establish a bit of terminology and introduce an example application which we will use in the following.

1.1 Systems and their Users

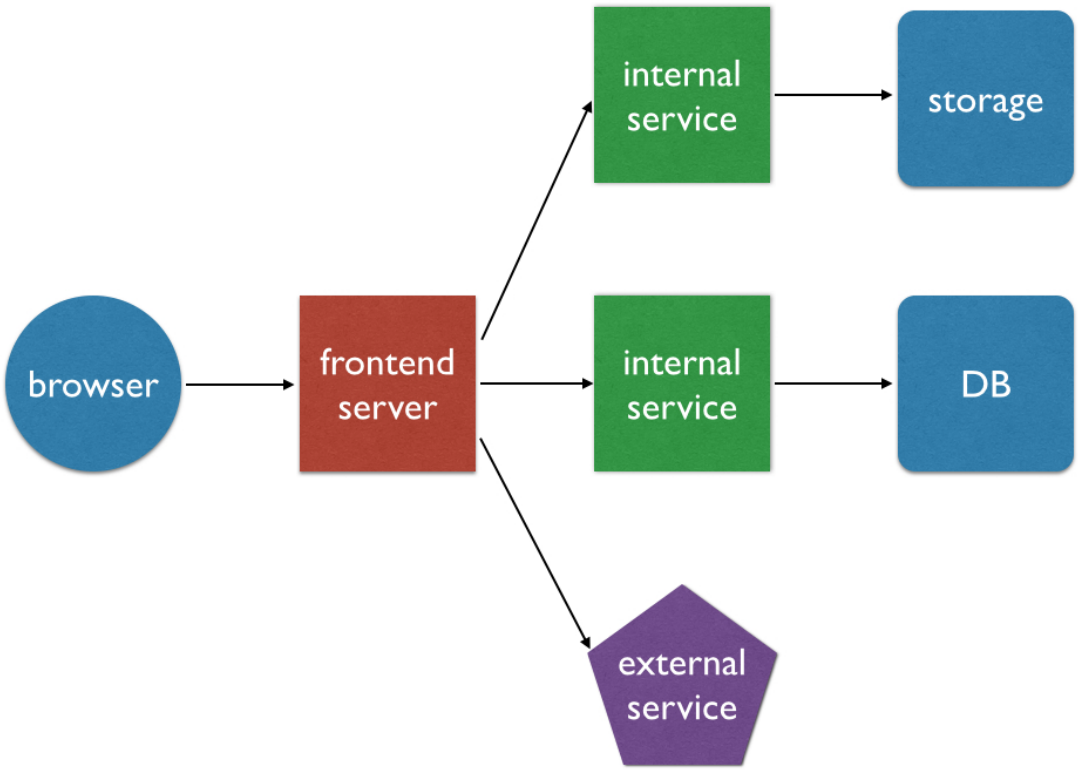

Figure 1.1 Schematic view of a typical web application deployment with different back-end services: the front-end server dispatches to different internal services, which in turn make use of other services like storage back-ends or databases.

In the paragraphs above we have used the word “user” informally and mostly in the

sense of humans who interact with a computer. This is without a doubt a very important aspect, but we can generalize this: Figure 1.1depicts a typical deployment of a web application. As an example, consider the GMail web application. 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 who is acting on behalf of a human, either directly or indirectly.

The first layer of services is provided by the frontend server and consumed by the web browser. The browser makes requests and expects responses—predominantly using HTTP, but also via WebSockets, SPDY, etc. The resources which are requested can pertain to emails, contacts, chats, searching and many more (plus the definition of the styles and layout of the web site). One such request might be related to the images for people you correspond with: when you hover over an email address, a pop-up window appears which contains details about that person, including a photograph or avatar image. In order to render that image, the web browser will make a request to the frontend server. The name of that server is not chosen without reason, because its main function is to hide all the backend services behind a façade so that external clients do not need to concern themselves with the internal structure of the overall GMail service implementation and Google is free to change it behind the scenes. The frontend server will make a request to an internal service for retrieving that person's image; the frontend server is thus a user of the internal service. The internal image service itself will probably contain all the logic for managing and accessing the images, but it will not contain the bits and bytes of the images itself, those will be stored on some distributed file system or other storage system. In order to fulfil the request, the image service will therefore employ the services of that storage system.

In the end, the user action of hovering the mouse pointer over an email address sets in motion a flurry of requests via the web browser, the frontend server, 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 we have seen multiple user–service relationships, and all of them need to meet the basic challenges as outlined in the introduction; most important is the requirement to respond quickly to each request. When describing reactive systems, we mean all of these relationships:

- a *user* which consumes a *service*

- a *client* which makes a request to a *server*
- a *consumer* which contacts a *provider*
- and so on

A system will comprise many parts that act as services, as shown above, and most of these will in turn be users of other services. The important point to note is that today's systems are distributed at an increasingly fine-grained level, introducing this internal structure in more and more places. When designing the overall implementation of a feature like the image service, you will 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, we need to turn our focus to making these services as responsive as they need to be to satisfy their users at all levels. Along the way we will see that lowering the granularity of our services allows us to better control, compose and evolve them.

1.2 Reacting to Users

The first and foremost quality of a service is that it must respond to requests it receives. This is quite obvious once you think about it: when you send an email via GMail, you want confirmation that it has been sent; when you select the label for important email then you want to see all important emails; when you delete an email you want to see it vanish from the displayed list. All of these are manifestations of the service's responses, rendered by the web browser to visualize them.

The same holds true in those cases where the user is not a human, because the services which consume other services expect responses as well in order to be able to continue to perform their function. And the users of these services also expect them to respond in a timely fashion in turn.

1.2.1 Responsiveness in a Synchronous System

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 faculty of 42. Whatever the function does, we 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 replying with a response. What makes this example so simple is that most programming languages include syntax like the one above 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 at all, because it cannot 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.

As soon as a computation needs to be distributed among multiple processor cores or networked computers, this tightly-knit package falls apart. Waldo et al note[4] that the assumptions woven into local, synchronous method calls are broken in several ways as soon as network communication is involved. The main problems are:

Footnote 4 Jim Waldo, Geoff Wyant, Ann Wollrath, Sam Kendall: A Note on Distributed Computing, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.7628

- vastly different latency between local and remote calls
- different memory visibility for local and remote calls
- the possibility for partial failure during remote calls
- inherent concurrency when performing remote calls

The Waldo article was written in 1994, when there was a clear hierarchy of latencies and bandwidths between local memory access, persistent storage and network communication. Within the last twenty years these lines have become blurred due to extremely fast network equipment on the one hand and rather costly inter-socket communications (between cores which are part of different processors in the same computer) on the other hand; instead of being separated by three or more orders of magnitude (sub-microsecond versus milliseconds) they have come within a factor ten of each other. We need to treat the invocation of services that are remote in the same manner as those running on the same machine but within different threads or processes, as most or all of the characteristic differences given above apply to them as well. The classical “local” way of formulating our systems is being replaced by designs which are distributed on ever more fine-grained levels.

1.2.2 Why is responsiveness now more important than ever?

The most costly problem with distributed systems is their capability of *partial failure*, which means that in addition to a failure within the target service we need to consider the possibility that either the request or the response might be lost; this may occur randomly, making the observed behavior seem inconsistent over time. In a computer network this is easy to see, just unplug a network cable and the packets transporting these messages will not be delivered. In the case of an internally distributed system you will have to employ different utilities for transporting requests and responses between different CPU cores or threads. One way to do this is to use queues which are filled by one thread and emptied by another[5]; if the former is faster than the latter then eventually the queue will run full (or the system will run out of memory), leading to message loss just as in a networked system. An alternative would be synchronous handover from one thread to another, where either is blocked from further progress until the other is ready as well. Such schemes also include faults which amount to partial failure due to the concurrent nature of execution—simply spoken the caller will not see the exception of the callee anymore since they do not run on the same thread.

Footnote 5 An example of such an architecture is the Staged event-driven architecture (SEDA), http://www.eecs.harvard.edu/~mdw/proj/seda/, about which an interesting retrospective is available at http://matt-welsh.blogspot.com/2010/07/retrospective-on-seda.html.

In a distributed system responsiveness therefore is not only a tunable property which is nice to have, it is a crucial requirement. The only way to detect that a request may not have been processed is to wait for the response so long that under normal circumstances it should have arrived already. But this requires that the maximum time between request and response—the maximum *response latency*[6]—is known to the user, and that the latency of the service is consistent enough for this purpose.

Footnote 6 Latency describes the time that passes between a stimulus and a reaction: when a physician knocks the rubber hammer against that particular spot below the knee your lower leg will jerk forward, and the time between when the hammer hits and when the leg starts moving is the latency. When you send an HTTP request, the response latency (at the application level) is the time between when you invoke the method that sends the request and when the response is available to you; this is greater than the processing latency at the server, which is measured between when the request is received and when the response is being sent.

For example consider the GMail app's contact image service: if it normally takes 50 milliseconds to return the bits and bytes, but in some cases it might take up to 5 seconds, then the wait time required by users of this service would need to

be derived from the 5 seconds, and not from the 50 milliseconds (usually adding some padding to account for variable network and process scheduling latencies). The difference between these two is that in one case the frontend could reply with a 503 “Service temporarily unavailable” error code for example after 100 milliseconds while in the other it would have to wait and keep the connection open for many seconds. In the first case the human user could see the generic replacement image with a delay that is barely noticeable while in the second case even a patient user will begin wondering if their internet connection is broken.

The distinguishing factor between the two scenarios is the vastly different limit defining the maximum reasonable wait time for a response. In order to obtain a reactive system we want this time to be as short as possible in order to live up to the expectation that responses are returned “quickly”. A more scientific formulation for this is that a service needs to establish an upper bound on its response latency which allows users to cap their wait times accordingly.

Important: In order to recognize when a request has failed we need bounded latency, which means formulating a dependable budget for which latency is allowed. Having a soft limit based on a latency which is “usually low enough” does not help in this regard. Therefore the relevant quantity is not the average or median latency, since the latter for example would mean that in 50% of the cases the service will still respond after the budget elapses, rejecting half of the valid replies due to timeout. When characterizing latencies we might look at the 99th percentile (i.e. the latency bound which allows 99% of the replies through and only rejects 1% of valid replies) or depending on the requirements for the service even on the 999th 1000-quantile—or even the 9999th 10000-quantile.

1.2.3 Cutting Down Latency by Parallelization

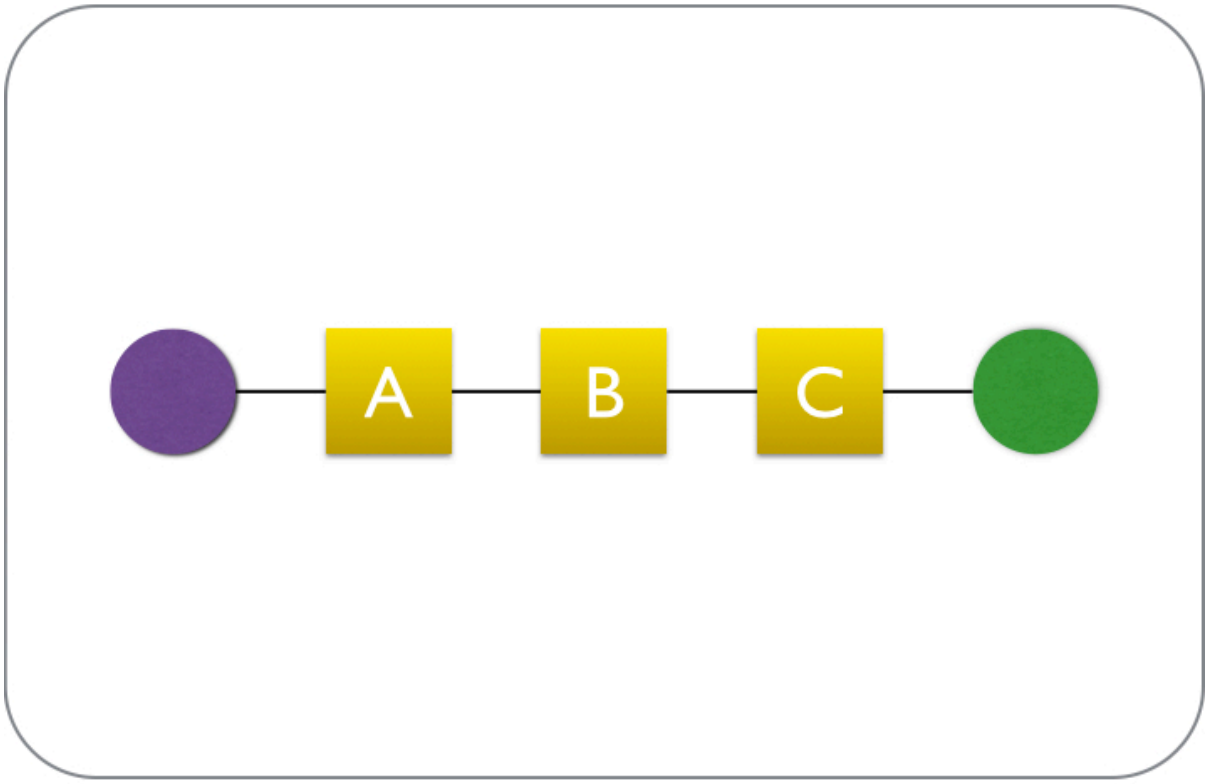

Figure 1.2 A task consisting of three sub-tasks that are executed sequentially: the total response latency is given by the sum of the three individual latencies.

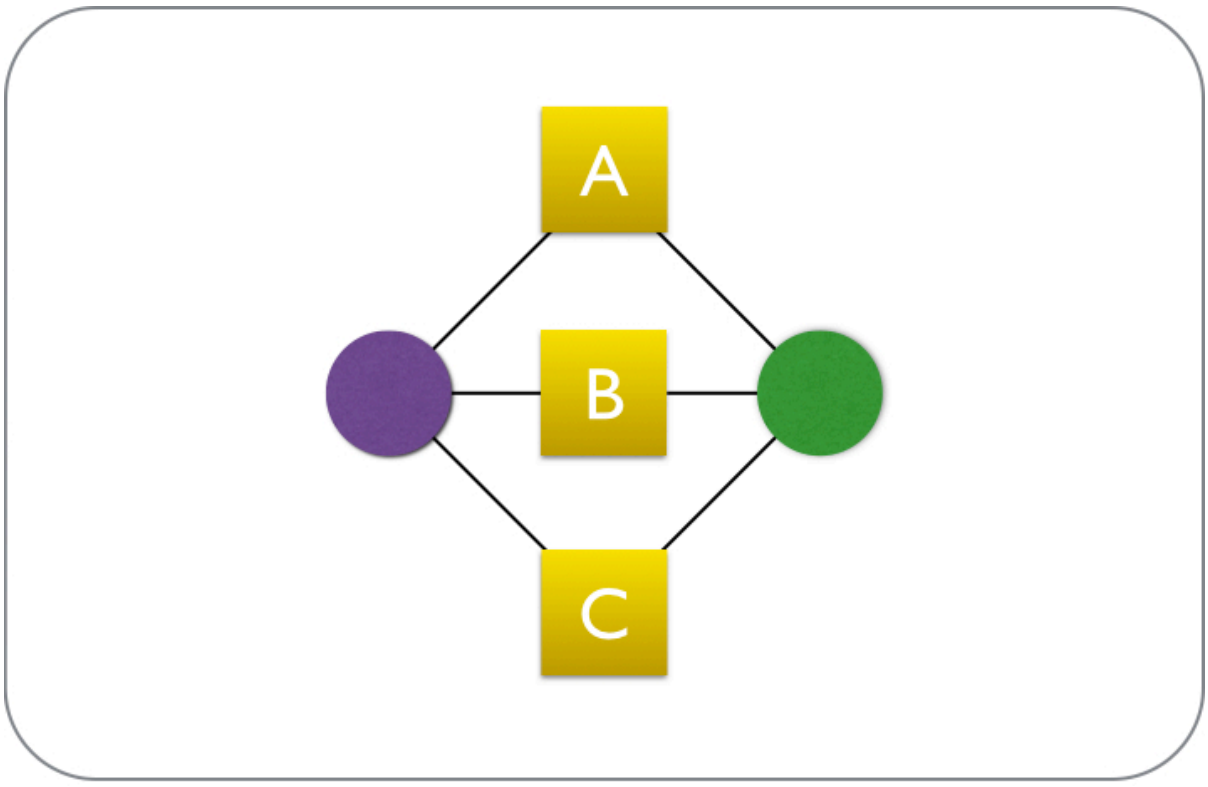

Figure 1.3 A task consisting of three sub-tasks that are executed in parallel: the total response latency is given by the maximum of the three individual latencies.

In many cases there is one possibility for latency reduction which immediately presents itself. If for the completion of a request several other services must be involved, then the overall result will be obtained quicker if the other services can perform their functions in parallel as shown in figure 1.3 above. This requires that no dependency exists such that for example task B needs the output of task A as one of its inputs, but that is frequently 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 contains textual information about that person as well as their image, and these can clearly be obtained in parallel.

When performing sub-tasks A, B and C sequentially as shown in figure 1.2 the overall latency will depend on the sum of the three individual latencies, while in the parallel case this part will be replaced with the latency of whichever of the sub-tasks takes longest. In this example we have just three sub-tasks, but in real social networks this number can easily exceed 100, rendering sequential execution entirely impractical.

Sequential execution of functions is well-supported by all popular programming languages out of the box:

```
// Java syntax
ReplyA a = taskA();
ReplyB b = taskB();
ReplyC c = taskC();
Result r = aggregate(a, b, c);
```

Parallel execution usually needs some extra thought and library support. For one, the service being called must not return the response directly from the method call which initiated the request, because in that case the caller would be unable to do anything while task A is 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:

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

This and other tools of the trade are discussed in detail in chapter two, here it suffices to know that a Future is a placeholder for a value which may eventually become available, and when it does the value can be accessed via the Future object. If the methods invoking sub-tasks A, B and C are changed in this fashion then the overall task just needs to call them to get back one Future each.

The code above uses a type called `Future` defined in the Java standard library (in package java.util.concurrent), and the only method it defines for accessing the value is the blocking `get()` method. Blocking here means that 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 I get the task to assemble the overview file of a certain client, I will dispatch three runners: one to the client archives to fetch address, photograph and contract status, one to the library to fetch all articles the client has written and one to the postal 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 for the runners to place their findings in the envelope and the last one to come back dispatches 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.

What the thread processing the request should do is to just describe how the values shall be composed to form the final result instead of waiting idly. This is possible with *composable Futures*, which are part of many other programming languages or libraries, including newer versions of Java (`CompletableFuture` is introduced in JDK 8). What this achieves is that the architecture turns completely from synchronous and blocking to asynchronous and non-blocking, where the underlying machinery needs to become *event-driven* in order to support this. The example from above would transform into the following[7]:

Footnote 7 This would also be possible with 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`.

```
// using Scala syntax
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 sub-task as well as its completion are just events which are raised by one part of the program and reacted to in another part, for example by registering an action to be taken when a Future is completed with its value. In this fashion the latency of the method call for the overall task does not even include the latencies for sub-tasks A, B and C. The system is free to handle other requests while those are being processed, reacting eventually to their completion and sending the overall response back to the original user.

Now you might be wondering why this second part of 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. Since parallel execution of the sub-tasks A, B and C depended on their initiating methods to return Futures instead of strict results, this

must also apply to the overall task itself. It very likely is 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 is paired with asynchronous and event-driven result aggregation. An added benefit is that additional events like task timeouts can be added without much hassle, since the whole infrastructure is already there: it is entirely reasonable to perform task A and couple the resulting future with one which holds a `TimeoutException` after 100 milliseconds and use that in the following. Then either of the two events—completion of A or the timeout—triggers those actions which were attached to the completion of the combined future.

1.2.4 Choosing the Right Algorithms for Consistent Latency

Parallelization can only reduce the latency to match that of the slowest code path through your service method: if you can perform A, B and C in parallel and then you need to wait for all three results before you can initiate D, E, F and G in order to be able to assemble the final result, then the latency of this whole process will be the sum of

- the maximum latency of A, B and C
- the maximum latency of D, E, F and G
- plus what the aggregation logic itself consumes
- plus some overhead for getting the asynchronous actions to run

Apart from the last point, every contribution to this sum can be optimized individually in order to reduce the overall latency. The question is only what precisely we should be optimizing for: when given a problem like sorting a list we can pick an algorithm from a library of good alternatives, instinctively reaching for that one which has its sweet spot close to where the input data are distributed. This usual optimization goal is not focused on latency but on performance, we consider the average throughput of the component instead of asking which of the choices provides the best “worst case” latency.

This difference does not sound important, since higher average throughput implies lower average latency. The problem with this viewpoint is that in the case of latency the average is almost irrelevant: as we have seen in the example with the GMail app above we need to cap the wait time in case something goes wrong, which requires that the nominal latency is strictly bounded to some value, and the service will be considered failed if it takes longer than the allotted time.

Considering the service to be failing when it is actually working normally should ideally not ever happen: we want failure detection to be as reliable as possible because otherwise we will be wasting resources. Given the following data, where would you position the cut-off?

- In 10% of all cases the response arrives in less than 10ms.
- In 50% of all cases the response arrives in less than 20ms.
- In 90% of all cases the response arrives in less than 25ms.
- In 99% of all cases the response arrives in less than 50ms.
- In 99.9% of all cases the response arrives in less than 120ms.
- In 99.99% of all cases the response arrives in less than 470ms.
- In 99.999% of all cases the response arrives in less than 1340ms.
- The largest latency ever observed was 3 minutes and 16 seconds.

Clearly it is not a good idea to wait 3 minutes and 16 seconds hoping that this will never accuse the service wrongly of failure, because this measurement was just largest observed one and there is no guarantee that longer times are impossible. On the other hand cutting at 50ms sounds nice (it is an attractively low number!), but that would mean that statistically 1 in 100 requests will fail even though the service was actually working. When you call a method, how often do you think it should fail without anything actually going wrong? This is clearly a misleading question, but it serves to highlight that choosing the right latency bound will always be a trade-off between responsiveness and reliability.

There is something we can do to make that trade-off less problematic. If the implementation of the service is chosen such that it does not focus on “best case” performance but instead achieves a latency distribution which is largely independent of the input data, then the latency bounds corresponding to the different percentiles will be rather close to each other, meaning that it does not cost much time to increase reliability from 99.9% to 99.99%.

Keeping the response latency independent from the request details is an important aspect of choosing the right algorithm, but there is another characteristic which is equally important and which can be harder to appreciate and take into account. If the service keeps track of the different users making requests to it or stores a number of items which grows with more intense usage, then chances are that the processing of each single request takes longer the more state the service has acquired. This sounds natural—the service might have to extract the response from an ever growing pile of data—but it also places a limit at how intensely the service can be used before it starts violating its latency bound too frequently to be

useful. We will consider this limitation in detail when discussing scalability later in this chapter, here it suffices to say that when faced with a choice of different algorithms you should not trade performance in the case of heavy service use for achieving lower latency in the case of a lightly used service.

1.2.5 Bounding Latency even when Things go Wrong

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

WITHIN THE SERVICE: USE BOUNDED QUEUES

When a service receives requests faster than it can handle them—when the incoming request rate exceeds the service's capacity—then these requests will have to queue up somewhere. There are many places in which queueing occurs without being visible at the surface, for example in the TCP buffers of the operating system kernel or in the currently suspended threads which await their next time slot for execution on a CPU core. In addition, an event-driven system usually has a mechanism which enqueues events for later processing, because if it did not—meaning that it executes all actions on the spot—it will run out of stack space for deeply nested action handlers[8]. All these queues have the purpose of transporting information from one execution context to another, and all of them have the side-effect of delaying that transfer for a period of time which is proportional to the current queue size.

Footnote 8 This would lead for example to a StackOverflowError on the JVM or segmentation violations in native code.

Another way of expressing this is that queueing requests up in front of a service has the effect of incurring additional latency for each of the requests. We can estimate this time by dividing the size of the queue which has built up by the average rate at which requests can be taken in. As an example consider a service which can process 1000 requests per second and which receives 1100 requests per second for some time. After the first second, 100 requests will have queued up, leading to an extra 100/(1000/s)=0.1s of latency. One second later the additional latency will be 0.2s and so on, and this number will keep rising unless the incoming onslaught of requests slows down.

When planning the service this extra delay must be taken into account, because

otherwise the latency bound will be violated exactly at the wrong time—when your service is used by more users, presumably because you were just about to be successful. The difficulty with taking this into account lies in the dynamic nature of the queue: how can we estimate the maximum reasonable size to factor into the calculation?

There is only one way to do this reliably, and that is to reduce the dynamic part of the equation as far as possible by minimizing the implicit queues and replacing them with explicit, deliberately placed queues within your control. For example the kernel's TCP buffers should be rather small and the network part of the application should focus on getting the requests from the network into the application's queue as fast as possible. Then the size of that queue can be tuned to meet the latency requirements by setting it such that it holds only as many elements as extra latency was allowed in the timing budget.

When new requests arrive while the buffer is full, then those requests cannot be serviced within the allotted time in any case, and therefore it is best to send back a failure notice right away, with minimum delay. Another possibility could be to route the overflow requests to a variant of the service which gives less accurate or less detailed answers, allowing quality of service to degrade gracefully and in a controlled fashion instead of violating the latency bound or failing completely.

USERS OF THE SERVICE: USE CIRCUIT BREAKERS

When users are momentarily overwhelming a service, then its response latency will rise and eventually it will start failing. The users will receive their responses with more delay, which in turn increases their own latency until they get close to their own limits. In order to stop this effect from propagating across the whole chain of user–service relationships, the 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 1.4.

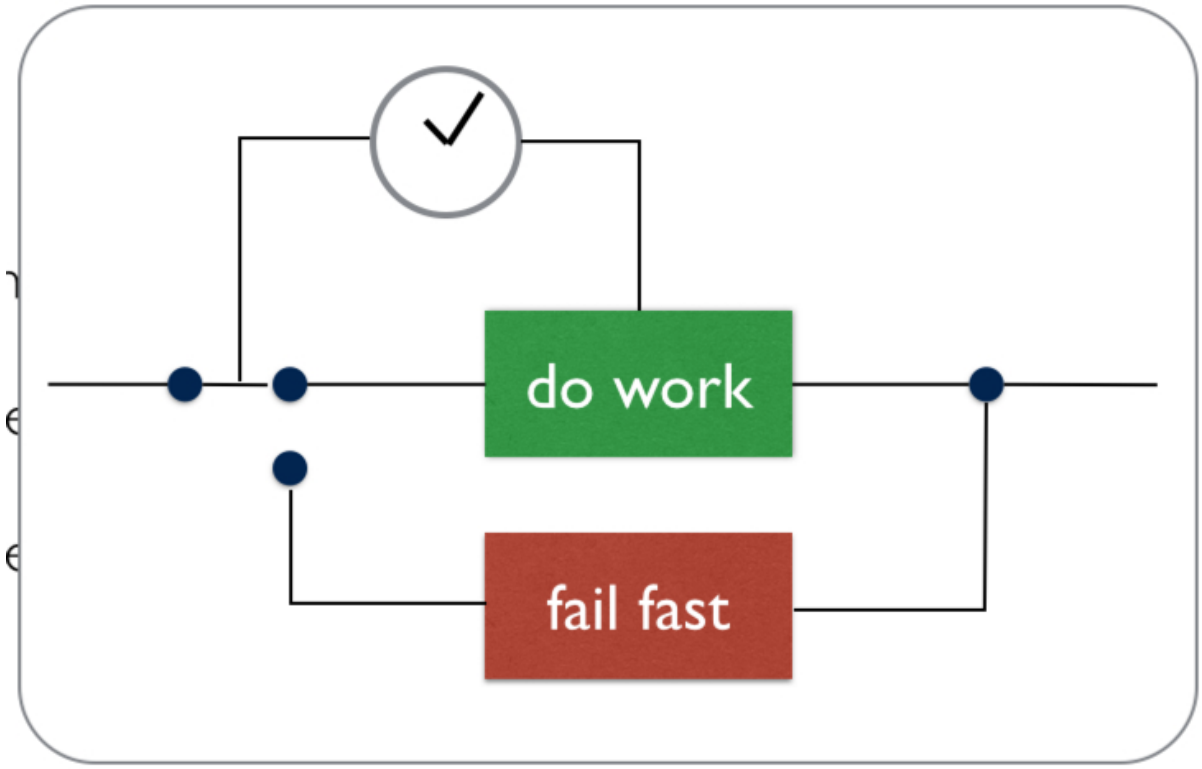

Figure 1.4 A circuit breaker in electrical engineering protects a circuit from being destroyed by a too high current. The software equivalent does the same thing for a service which would otherwise be overwhelmed by too many requests.

The idea here is quite simple: when involving another service, monitor the time it takes until the response comes back. If the time consistently rises above the allowed threshold which this user has factored into its own latency budget for this particular service call, then the circuit breaker trips and from then on requests will take a different route of processing which 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 at all.

This does not only benefit the user by insulating it from the faulty service, it 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 a possibility to monitor such occurrences and reinforce the resources for the overwhelmed service in response to the increased load, as we shall discuss below when we talk about scalability.

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 try out whether the service is back in shape. If not, then it can trip immediately again, otherwise it closes automatically and resumes normal operations.

1.2.6 Summarizing the Why and How of Responsiveness

The top priority of a service is that it responds to requests in a timely fashion, which means that the service implementation must be designed to respect a certain maximum response time—its latency must be bounded. Tools available for achieving this are

- making good use of opportunities for parallelization
- focusing on consistent latency when choosing algorithms
- employing explicit and bounded queuing within the service
- shielding users from overwhelmed services using circuit breakers

1.3 Reacting to Failure

In the last section we concerned ourselves with 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 or not. But even with flawless execution of this design unexpected things will happen eventually:

- *Software will fail.*
  There will always be that exception which you forgot to handle (or which was not even documented by the library you are using), or you get synchronization only a tiny bit wrong and a deadlock occurs, or that condition you formulated for breaking that loop just does not cope with that weird edge case. You can always trust the users of your code to figure out ways to find all these failure conditions and more.
- *Hardware will fail.*
  Everyone who has operated computing hardware knows that power supplies are notoriously unreliable, or that harddisks tend to turn into expensive door stops either during the initial burn-in phase or after a few years later, or that dying fans lead to 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 tasking maintenance personnel with replacing that failed harddisk in the RAID5, a study[9] finds that there is a 10% chance to replace the wrong one, leading to the loss of all data. An anecdote from Roland's days as 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 should happen, but it is human nature that we just have a bad day from time to time.

Footnote 9 Aaron B. Brown, IBM Research, Oops! Coping with Human Error in IT Systems, http://queue.acm.org/detail.cfm?id=1036497

The question is therefore 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 it will get is that no normal response is received. It might be that connections time out or are rejected, or that the response consists 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 and the booking site stops responding then you will take your business elsewhere and probably not come back anytime soon.

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

NOTE **What does Resilience mean?**

Merriam-Webster defines resilience as:

- 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 we know that the latter will not be fully successful. It is of course good to plan for as many failure scenarios as we can, to tailor programmatic responses such that normal operations can be resumed as quickly as possible—ideally without the user noticing anything. But the same must also apply to those failure cases which were not foreseen explicitly in the design, knowing that these will happen as well.

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 eggs in one basket”, while the latter adds “protect your baskets from one another”. When it comes to handling the failure, it is important to *delegate*, so that not the failed compartment itself is responsible for its own recovery.

Distribution can take several forms, the one you think of first is probably that an important database is replicated 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 might go as far as placing the replicas

in different buildings in order not to lose all of them in case of a 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 even in different countries or on separate continents.

1.3.1 Compartmentalization and Bulkheading

The further apart the replicas are kept the smaller is the probability of a single fault affecting all of them. This can also be applied to the human component of the design, where operating parts of the system by different teams minimizes chances that the same mistake is made everywhere at once. The idea behind this is to isolate the distributed parts, or to use a metaphor from ship building we want to use *bulkheading*.

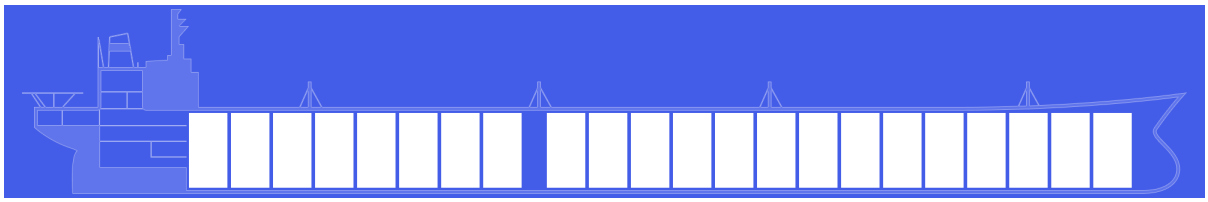

Figure 1.5 The term “bulkheading” comes from ship building and means that the vessel is segmented into fully isolated compartments.

Figure 1.5 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, then only those compartments which are directly affected will fill up with water and the others will remain properly sealed, keeping the ship afloat.

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[10]. We all know that that particular ship did in fact sink, so what went wrong? In order not to inconvenience passengers (in particular in 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.

Footnote 10 "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

This example—while 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 actually isolated from each other, failure can cascade between them to bring the whole system down. One such example from distributed computing designs is managing fault-tolerance at the level of whole 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 them unsinkable, but in order to obtain the catastrophic outcome the ship needs to be mismanaged severely and run with full speed against a rock. That metaphor translates in full to computer systems.

1.3.2 Consequences of Distribution

Executing different replicas of a service or entirely different services in a distributed fashion means that requests will have to be sent to remote computers and responses will have to travel back. This could be done by making a TCP connection and sending the request and response in serialized form, or it could use some other network protocol. The main effect of such distribution is that the processing of a request happens asynchronously, outside of the control of the user.

As we have seen when discussing the parallelization of tasks, asynchronous execution is best coupled with event-driven reply handling, since we will otherwise have one thread idly twiddling its thumbs while waiting for the response to come back. This is even more important when a network is involved, since the latency will in general be higher and the possibility of message loss needs to be taken into account, thus the effect of (partial) failure on the calling party in terms of wasted resources will be larger[11].

Footnote 11 This is even more relevant if for example a new TCP connection needs to be established, which adds overhead for the three-way handshake and in addition throttles the utilized bandwidth initially due to its slow-start feature.

For this reason distribution naturally leads to a fully asynchronous and event-driven design. This conclusion also follows from the principle of compartmentalization, since sending the request synchronously and processing it within the same method call holds the user's thread hostage, unable to react to further events during this time. Avoiding this means that processing must happen asynchronously and the response will be a future event which needs to be handled when it occurs; if the user blocked out all other inputs while waiting for the response then this scheme would not be an improvement over synchronous processing:

```
// Java syntax
Future<Reply> futureReply = makeRequest();
Reply reply = futureReply.get();
```

In this example the processing of the request happens asynchronously—perhaps on another computer—and the calling context can in principle go on to perform other tasks. But it chooses to just wait for the result of processing the requests, putting its own liveness into the hands of the other service in hope that it will reply quickly. Full isolation means that no compartment can have such power over another.

1.3.3 Delegating Failure Handling

The design we have constructed at this point consists of services which make use of other services in a way that completely isolates them from each other concerning failures in order to avoid those failures from cascading across the whole system. But as we have argued failure will eventually happen, so the question becomes where it should go. In traditional systems composed using synchronous method calls the mechanism for signaling and handling failure is provided by exceptions: the failing service throws an exception and the user catches and handles it. This is impossible in our isolated design since everything will need to be asynchronous, processing will happen outside of the call stack of the user and therefore exceptions cannot reach it.

THE VENDING MACHINE

At this point, let’s step back from designing computer systems and think about an entirely mundane situation: you need a cup of coffee and therefore make your way to the vending machine. While walking over you begin sorting through the coins in your pocket, so that you can immediately put the appropriate ones into the coin slot when you arrive. Thereafter you will press the right button and wait for the machine to do the rest. There are several possible outcomes to this operation:

- Everything went just fine and you can enjoy your steaming hot coffee.
- You mixed up the coins while day-dreaming and the machine points that out to you.
- The machine has run out of coffee and spews the coins back out.

All of these are perfectly reasonable exchanges between you and the machine, each giving a response paired with your request. The frustratingly negative outcomes listed above are not failures, they are nominal conditions signaled

appropriately by the machine; we would call those validation errors returned from the service.

Failures on the other hand are characterized by the inability of the machine to give you a response. Among the possible scenarios are:

- The logic board in the machine is broken.
- Some jerk stuck chewing gum into the coin slot so that the coins do not reach the counting mechanism.
- The machine is not powered.

In these cases there will not be a response. Instead of waiting indefinitely in front of it you will quickly realize that something is wrong, but what do you do? Do you fetch screwdriver and soldering iron and start fixing the machine? In all but the most special cases this is not what is going to happen, since you will simply walk away, hoping to get your shot of caffein in some other way. Maybe you inform someone of the failure, but in most cases you will just assume that those who operate the vending machine will eventually fix it—they will not earn any money with it while it is broken in any case.

SUPERVISION

The crucial observation in the example of the vending machine is that responses—including validation errors—are communicated back to the user of a service while failures must be handled by the one who operates the service. The term which describes this relationship in a computer system is that of a *supervisor*. The supervisor is responsible for keeping the service alive and running.

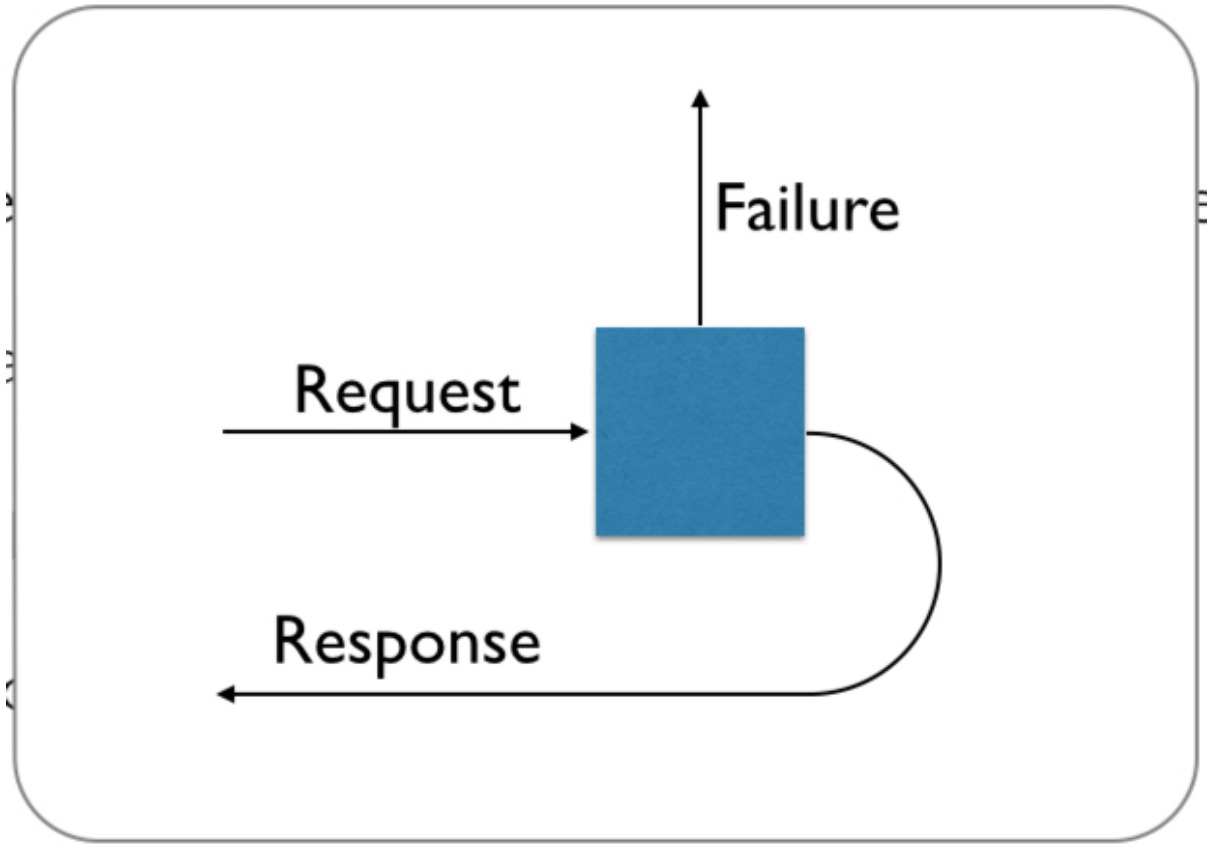

Figure 1.6 Supervision means that normal requests and responses (including negative ones such as validation errors) flow separately from failures: while the former are exchanged between the user and the service, the latter travel from the service to its supervisor.

Figure 1.6 depicts these two different flows of information. The service internally handles everything which it knows how to, it performs validation and processes requests, but any exceptions which 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 which depends on a working database connection. When the connection breaks, the database driver will throw an exception. If we tried to handle this 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 is that 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 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 backend system and fail over to a secondary database replica when appropriate. In order to do that the supervisor must have the power to start, restart and stop the services it supervises, it is responsible for their lifecycle.

The first system which directly supported this concept was Erlang/OTP, implementing the Actor model which is discussed in chapter two. Patterns related to supervision are described in chapter five.

1.3.4 Summarizing the Why and How of Resilience

The user of a service expects responsiveness at all times and typically has not much tolerance for outages due to internal failure. The only way to make a service resilient to failure is to distribute and compartmentalize it:

- Install water-tight bulkheads between compartments to isolate failure.
- Delegate handling of failure to a supervisor instead of burdening the user.

1.4 Reacting to Load

Let us assume that you built a service which is responsive and resilient and you offer this service to the general public. If the function that the service performs is also useful and possibly en vogue then eventually the masses will find out about it. Some of the early adopters will write blog posts and status updates in social networks and some day such an article will end up on a large news media site and thousands of people suddenly want to know what all the noise is about. In order to save cost—you have invested your last penny into the venture but still need to buy food occasionally—you run the site using some infrastructure provider and pay for a few virtual CPUs and a little memory. The deployment will not withstand the onslaught of users and your latency bounds kick in, resulting in a few hundred happy customers[12] and a few thousand unimpressed rubbernecks who will not even remember the product name you carefully made up—or worse they badmouth your service as unreliable.

Footnote 12 This is an interesting but in this case not really consoling win of properly designing a responsive system; with a traditional approach everyone would have just seen the service crash and burn.

This sad story has happened many times on the internet to date, but it lies in the nature of this kind of failure that it is usually not noticed. A very famous example of a service which nearly shared this fate is Twitter. The initial implementation was simply not able to keep up with the tremendous growth that the service experienced in 2008, and the solution was to rewrite the whole service over the

period of several years to make it fully reactive. Twitter had at that point already enough resources to turn impending failure into great success, but most tiny start-ups do not.

How can you avoid this problem? How can we design and implement a service such that it becomes resilient to overload? The answer is that the service needs to be built from the ground up to be scalable. We have seen that distribution is necessary in order to achieve resilience, but it is clear that there is a second component to it: a single computer will always have a certain maximum number of requests that it can handle per second and the only way to increase capacity beyond this point is to distribute the service across multiple computers.

The mechanics necessary for distribution are very similar to those we discussed for parallelization already. The most important prerequisite for parallelization was that the overall computation can be split up into independent tasks such that their execution can happen asynchronously while the main flow goes on to do something else. In the case of distribution of a service in order to utilize more resources we need to be able to split up incoming stream of work items (the requests) into multiple streams that are processed by different machines in parallel.

As an example consider a service which calculates mortgage conditions: when you go to a bank and ask for a loan then you will have to answer a number of questions and your answers determine how much you would have to pay per month and how much you still owe the bank after say 5 years. When the bank clerk presses the calculate button in the internal mortgage web application, a request is made to the calculation service which bundles all your answers. Since many people want to inquire about loans all the time all over the country, the bank will have an instance of this service running at headquarters which day in and day out crunches the numbers on all these loans. But since there is no relationship between any two such requests, the service is free to process them in any order or as many in parallel as it wants, the incoming work stream consists of completely independent items and is therefore splittable down to each single request.

Now consider the service within the same bank which handles the actual loan contracts. It receives very similar data with its requests, but the purpose is not an ephemeral calculation result. This service must take care to store the contract details for later reference, and it must also give reliable answers as to which loans a certain person already has taken. The work items for this service can be correlated

with each other if they pertain to the same person, which in turn means that the incoming request stream cannot be split up arbitrarily—it can still split up though and we will discuss techniques for that in chapter seven.

The second component to making a service scalable builds on top of splittable stream of work in that the performance of the service is monitored and the distribution across multiple service instances is steered in response to the measured load. When more requests arrive and the system reaches its capacity then more instances are started to relieve the others; when the number of requests decreases again instances are shut down to save cost. There is of course an upper limit to how far you can allow the system to scale, and when you hit that ceiling then there is no choice but to let the responsiveness measures kick in and reject requests—given today's cloud service providers this limitation can be pushed out quite far, though.

1.4.1 Determining Service Capacity

There is one very useful and equally intuitive formula relating the average number of requests L which are concurrently being serviced, the rate λ at which requests arrive and the average time W a request stays in the service while being processed:

$$L = \lambda \bullet W$$

This is known as *Little's Law* and it 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. As an example consider the case that servicing one request takes 10ms and only one request arrives per second on average. Little's Law tells us that the average number of concurrent requests will be 0.01, which means that when planning the deployment of that service we do not have to foresee multiple parallel tracks of execution. If we want to use that service more heavily and send it 1000 requests per second then the average number of concurrent requests will be 10, meaning that we need to foresee the capability to process ten requests in parallel if we want to keep up with the load on average.

We can use this law when planning the deployment of a service. To this end we must measure the time it takes to process one request and we must make an educated guess about the request frequency—in the case of internal services this number can potentially be known quite well in advance. The product of these two numbers then tells us how many service instances will be busy in parallel, assuming that they run at 100% capacity. Normally there is some variability expected in both the processing time and the request frequency, wherefore you will target something between 50–90% of average service utilization in order to plan for some headroom.

What happens during short-term periods where bursts of requests exceed the planned number has already been discussed under the topic of using bounded queues for bounded latency: the queues in front of your service will buffer a burst up to a certain maximum in extra latency and everything beyond that will receive failure notices.

1.4.2 Building an Elastic Service

Little's Law can also be used in a fully dynamic system at runtime. The initial deployment planning may have been guided by measurements, but those were carried out in a testbed with test users. Especially in case of a public service it is hard to come up with realistic test conditions since people browsing the internet can at times behave unpredictably. When that happens you will need to measure again and redo the math to estimate the necessary service capacity. Instead of waking up the system architect on a Sunday night it would be much preferable if this kind of intelligence were built into the service itself.

Given an event-driven system design we have everything that is needed in place already. It does not cost much to keep tabs on how long the processing of each request takes and feed this information to a monitoring service—the word *supervisor* comes to mind. The total inflow of requests can also easily be measured at the service entry point and regular statistics sent to the supervisor as well. With these data it is trivial to apply for example an exponentially decaying weighted average or a moving average and obtain the two input values to Little's formula. Taking into account the headroom to be added we arrive at

λ = average(requestsPerSecond)

W = average(processingTime)

u = targetUtilization

$L = \lambda \cdot W / u$

Upon every change of the input data this formula is evaluated to yield the currently estimated resource need in number of service instances, and when L deviates sufficiently[13] the supervisor changes the number of running instances automatically. The only danger in such a system is that the users indirectly control the resources spent on their behalf, which can mean that your infrastructure bill at the end of the month will be higher when your service is used a lot. But presumably that is a very nice problem to have since under this scheme your service should have won many satisfied customers and earned money accordingly.

Footnote 13 You will want to avoid reacting on pure fluctuations in order to keep operations smooth.

1.4.3 Summarizing the Why and How of Scalability

A successful service will need to be scalable both up and down in response to changes in the rate at which the service is used. Little's Law can be used to calculate the required deployment size given measurements for the request rate and the per-request processing time. In order to make use of it you must consider:

- The incoming request stream must be splittable for distribution.
- Request processing must be distributable and event-driven.
- Change number of active instances based on monitoring data at runtime.

1.5 Reacting to Events

All of the tenets of the Reactive Manifesto which we have considered so far lead to an event-driven design. We have encountered it when asynchronously aggregating the results of parallelized sub-tasks, it was implicit in the notion of queueing up requests in front of a service (and managing that queue to achieve bounded latency), it is a necessary companion of supervision in that the supervised service must communicate with users as well as supervisor (which necessitates handling incoming events as they occur), and we have seen how a service based on distributable events (requests as well as monitoring data) can dynamically scale up and down to adapt to changing load. But focusing on events instead of traditional method calls also has benefits on its own.

1.5.1 Loose Coupling

We have seen that resilience demands that we distribute a service and form compartments which are isolated from each other. This isolation at runtime cannot be achieved if the code paths describing each compartment are entangled with each other. If that were the case then parts of the code of one compartment would make assumptions on how exactly the other compartment's function is implemented and depend on details which should better be encapsulated and not exposed. The more independent the implementations are, the smaller is the probability of repeating the same mistake in several places, which could lead to cascading failures at runtime—a software failure in one compartment would happen in another as well.

The idea behind modeling encapsulated units which are executed in isolation and communicate only via their requests and responses is to decouple them not only at runtime, but also in their design. The effort of analysing and breaking down

the problem into actionable pieces will then focus purely on the communication protocols between these units, leading to a very strict separation of interface and implementation.

TODO: add figure about Alan Kay's objects.

The original purpose of object orientation as described by Alan Kay was to fully encapsulate state within the objects and describe their interactions as message protocols between them. If you look at Java EE objects from this angle, you can imagine that every method call you make is sending a message to the object, and the object responds with the result. But something is not quite right with this picture: sending a message and awaiting a reply are done by the user while processing the message and generating the reply should be done by the service. With Java objects every caller can force the object to process the request right there, independent of the state the object might be in. This makes it very convenient to burden the object with more and more methods—like getters and setters—which would normally not be considered if the object had the right to respond to requests in its own time, when it is appropriate. In other words normal Java objects do not exhibit the kind of encapsulation that Alan Kay talked about[14].

Footnote 14 It can well be argued that Java is not so much an object oriented language as it is a class oriented one: it focuses primarily on the inheritance relationships between types of objects instead of on the objects themselves.

If you take a step back and think about objects as if they were persons, each with their right to privacy, then it becomes very obvious how they should interact. Persons talk with each other, they exchange messages at a very high level. Calling setters on an object corresponds to micromanagement at the lowest possible level, it is as if you told another person when to breathe (including the consequences in case you forget to do so). Instead, we talk about what we will do the next day, week or year, and the realization of these messages will comprise a huge number of steps which each person executes autonomously. What we gain by this is encapsulation and we reduce the communication protocol to the minimum while still achieving the goals we have set.

Turning our view again onto objects we conclude that a design which focuses on the communication protocol, on events which are sent and received, will naturally lead to a higher level of abstraction and less micromanagement, because designing all those requests and responses at too low level would be tedious. The

result will be that independently executing compartments will also be largely independent in the source code. The benefits of such an architecture are self-evident:

- Components can be tested and verified in isolation.
- Their implementation can be evolved without affecting their users.
- New functionality corresponds to new communication protocols, which are added in a conscious fashion.
- Overall maintenance cost is lower.

1.5.2 Better Resource Utilization

Loose coupling between components—by design as well as at runtime—includes another benefit: more efficient execution. Modern hardware does not advance any more primarily by increasing the computing power of a single sequential execution core, physical limits[15] have started impeding our progress on this front around the year 2006 so our processors host more and more cores instead. In order to benefit from this kind of growth we must distribute computation even within a single machine. 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.

Footnote 15 The speed of light as well as power dissipation make further increases in clock frequency impractical.

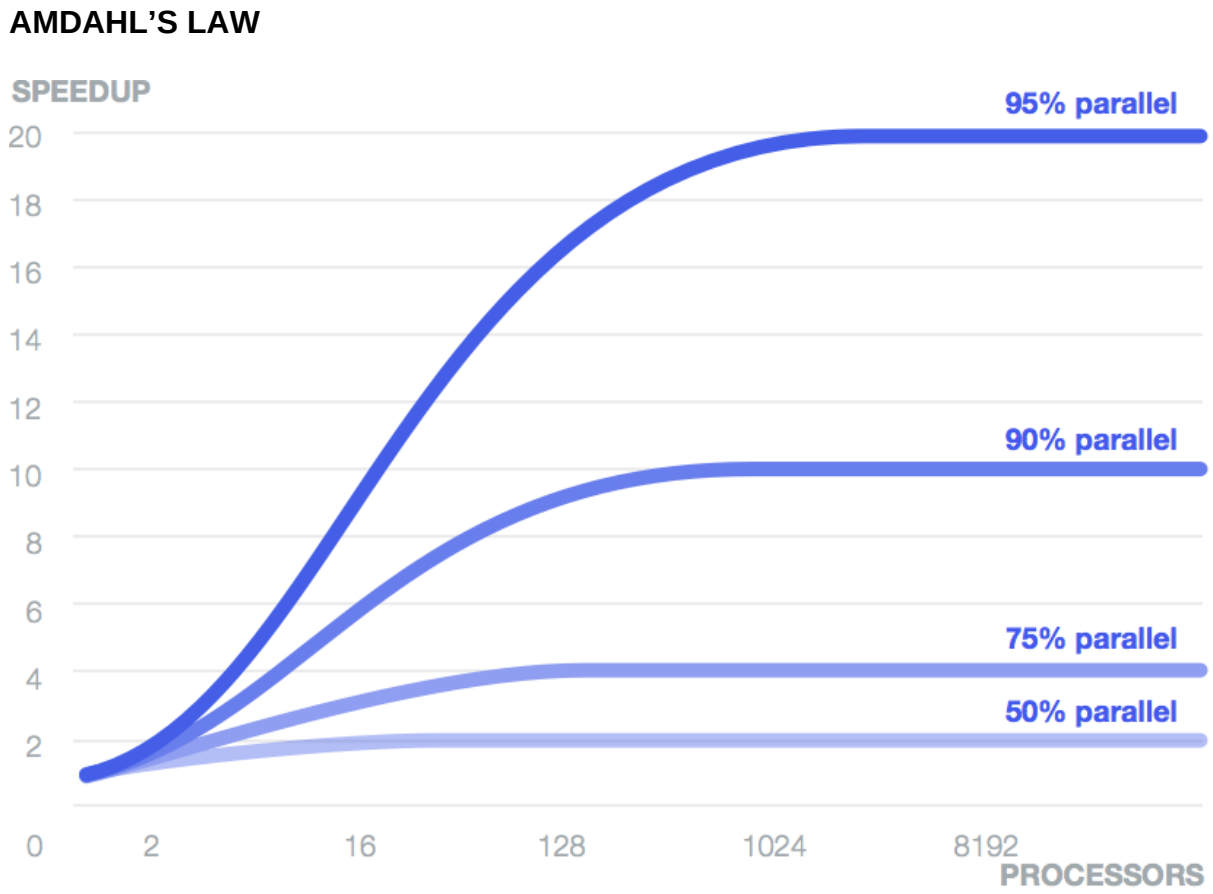

Figure 1.7 The speed-up of a program using multiple processors in parallel computing is limited by the sequential fraction of the program. For example if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20 times, no matter how many processors are used.

Coordinating the access to a shared resource—for example a map holding the state of your application indexed by username—means executing those portions of the code which depend on the integrity of the map in some synchronized fashion. This means that effects which change the map need to happen in a *serialized* fashion in some order which is globally agreed upon by all parts of the application; this is also called *sequential consistency*. There is an obvious drawback to such an approach: those portions which 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 which is achievable by parallelization is captured by Amdahl's Law:

$S(n)=T(1)/T(n)=1/(B+1/n(1-B))$

Here n is the number of available threads, B 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 the figure above for different values of *B* across a range of available threads—they translate into the number of CPU cores on a real system. If you look at it you will notice that even if only 5% of the program run inside these synchronized sections and the other 95% are parallelizable, the maximum achievable gain in execution time is a factor of 20, and getting close to that theoretical limit would mean employing the ridiculous number of about 1000 CPU cores.

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 is necessary—in which case scalability would be perfect: in the formula above *B* would be zero, simplifying the whole equation to

$$S(n)=n$$

In plain words this means that using *n* times as many computing resources we achieve *n* times the performance. If we build our system on fully isolated compartments which are executed independently, then this will be the only theoretical limit, assuming that we can split the task into at least *n* compartments. In practice we 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 events per second between CPU cores.

LOWER COST OF DORMANT COMPONENTS

Traditional ways to model interactions between components—like sending to and receiving from the network—are expressed as blocking API calls:

```
// e.g. using Java API
final Socket socket = ...
socket.getOutputStream.write(requestMessageBytes);
final int bytesRead = socket.getInputStream().read(responseBuffer);
```

Each of these interact with the network equipment, generating events and reacting to events under the hood, but this fact is completely hidden in order to construct a synchronous façade on top of the underlying event-driven system. This means that 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 which is ongoing in parallel needs its own thread even if many of these 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 would 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`, while the time for traversing the network and processing the request on the other end is measured in milliseconds, it is clear that each of these threads 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 we should note that threads are managed by the operating system kernel for efficiency reasons[16]. Since 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 *context switch* and costs thousands of cycles[17] every time it occurs. The other part of using large numbers of threads is that the scheduler—that part of the kernel which 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.

Footnote 16 Multiplexing several logical user-level threads on a single O/S thread is called an 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).

Footnote 17 While CPUs have gotten faster, their larger internal state negated the advances made in pure execution speed such that a context switch has taken roughly 1µs since over a decade.

The takeaway of the previous paragraph is that using synchronous, blocking APIs which hide the underlying event-driven structure waste CPU resources. If the events 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:

```
// using (remote) messaging between Akka actors from Java 8
```

```
Future<Response> future = ask(actorRef, request, timeout)
                          .mapTo(classTag(Response.class));
future.onSuccess(f(response -> /* process it */));
```

Here the sending of a request returns a handle to the possible future reply—a composable Future as discussed in chapter two—upon 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.

1.5.3 Summarizing the Why and How of Event Orientation

The main benefits of event-driven designs are:

- Event transmission between components allows a loosely coupled architecture with explicit protocols.
- "Share nothing" architecture removes scalability limits imposed by Amdahl's law.
- Components can remain inactive until an event arrives, freeing up resources.

In order to realize these, non-blocking and asynchronous APIs must be provided which explicitly expose the system's underlying event structure.

1.6 How does this Change the Way We Program?

The most consequential common theme of the tenets of the Reactive Manifesto is that distribution of services and their data is becoming the norm, and in addition the granularity at which this happens is becoming more fine-grained. Multiple threads or even processes on the same computer used to be regarded as "local", the most prominent assumption of which is that as long as the participants in a system are running they will be able to communicate reliably. As soon as network communication is involved we all know that communication dominates the design—both concerning latency and bandwidth as well as concerning failure modes.

With bulkheads between compartments that may fail in isolation and which communicate only by asynchronous messages, all such interactions need to be considered as distributed even if they just run on different cores of the same CPU—remember Amdahl's law and the resulting desire of minimizing synchronization.

1.6.1 The Loss of Strong Consistency

One of the most famous theoretical results on distributed systems is Eric Brewer's CAP theorem[18], which states that any networked shared-data system can have at most two of three desirable properties:

Footnote 18 S. Gilbert, N. Lynch, Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services, ACM SIGACT News, Volume 33 Issue 2 (2002), pg. 51-59, http://lpd.epfl.ch/sgilbert/pubs/BrewersConjecture-SigAct.pdf

- consistency (C) equivalent to having a single up-to-date copy of the data;
- high availability (A) of that data (for updates); and
- tolerance to network partitions (P).

This means that during a network partition we have to sacrifice at least one of consistency and availability. In other words if we continue allowing modifications to the data during a partition then inconsistencies can occur, and 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. The document is hopefully stored in at least two different locations 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, see their own changes but not the changes made by the respective 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 will not only be unable to make conflicting changes but they will also be prevented from working on completely unrelated parts of the document as well.

Traditional data stores are relational databases which 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[19] transaction semantics. For this reason distributed systems have so far concentrated critical components in such a way that provided strong consistency.

Footnote 19 Atomicity, Consistency, Isolation, Durability

In the example of the two users editing the shared document, a corresponding strongly consistent solution would mean that every change—every key press—would need to be confirmed by the central server before being displayed locally, since 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 very irritating to have such high latency while typing text, we are used to the characters appearing instantly. And this solution would also be quite 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 sections above 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.

While the time affected time window at the currently typical velocities with which computers travel is extremely small, another effect is that events which cannot be connected by a ray of light as described above 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 in order to avoid ambiguities, but this is becoming a painful restriction already within today's processor designs: agreeing on the the current clock tick on both ends of a silicon chip is one of the limiting factors when trying to increase the clock frequency.

Distributed systems therefore build on a different set of goals called BASE instead of the synchrony-based ACID:

- 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 replica, and during this time it is possible for external observers to see data which are inconsistent. The qualification "eventually" means that 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 that while 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 might be different. But once the incoming streams of changes ends both views will eventually settle into the same state for both users.

In a note[20] written twelve years after the CAP conjecture Eric Brewer remarks:

> 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.

Footnote 20 E. Brewer, CAP Twelve Years Later—How the "Rules" Have Changed, http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

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 rather short-lived; the delay between changes being made by one user and being visible to others is of the order of tens or maybe hundreds of milliseconds, which is good enough for collaborative document editing.

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 CRDTs[21]. These have the property of merging cleanly when the partition ends regardless of the modifications that were done on either side.

Footnote 21 Conflict-free Replicated Data Types

Google Docs employ a similar technique called Operational Transformation[22]. In the scenario that 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 will be transformed to operate on the state which 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.

Footnote 22 http://www.waveprotocol.org/whitepapers/operational-transform

Data types with these nice properties come with certain restriction in terms of which operations they can support. There will naturally be problems which cannot be stated using them, in which case one has no choice but to concentrate these data in one location only and forego 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 of strong consistency. One example of this kind from the real world are ATMs[23]: 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.

Footnote 23 Automated Teller Machine

When you go to an ATM to withdraw cash, you would be rather annoyed with your bank if the ATM did not work, especially if you need the money to buy that anniversary present for your spouse. Network problems do occur frequently, which means that 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 only get a smaller amount of cash while the machine cannot verify that your account has sufficient funds, but you still get some bills instead of a dire “Out of Service” error. For the bank this means that your account may have gone negative, but chances are that most peoples who want to withdraw money actually do have enough to cover this transaction. And if the account now turned into a mini loan then there are established means to fix that: society provides a judicial system to enforce those parts of the contract which the machine could not, and in addition the bank actually earns money by earning 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.

1.6.2 The Need for Reactive Design Patterns

Many of the discussed solutions and in fact 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 “Design Patterns” book by Gamma, Helm, Johnson and Vlissides, published in 1994. 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 into 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 our charted territory. In 2003 Hohpe and Woolf published their “Enterprise Integration Patterns” which cover message passing between networked components, defining communication and message handling patterns—for example implemented by the Apache Camel project. The next step was termed Service Oriented Architecture.

While reading this chapter you will have recognized elements of earlier stages, like the focus on message passing or on services. The question naturally arises what this book adds that has not already been described sufficiently elsewhere. Especially interesting is a comparison to the definition of SOA in Rotem-Gal-Oz’s “SOA Patterns”:

> *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 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 there are technical reasons which push the coarseness of services down to finer levels and demand that abstractions like synchronous blocking network communication are replaced by explicitly modeling the event-driven nature of the underlying system.

Lifting the level of abstraction has proven to be the most effective measure in increasing the productivity of programmers. Exposing more of the underlying details seems like a step backwards on this count, since abstraction is usually meant to hide complications from view and solving them once—and hopefully correctly—instead of making mistakes while solving it over and over again. What this consideration neglects is that there are two kinds of complexity:

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

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

A proper solution exposes exactly 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 which results from a mismatch between the chosen abstraction and the underlying mechanics.

This means that as our understanding of the problem domain evolves—for example recognizing the need for distribution of computation at much finer granularity than before—we need to keep re-evaluating 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 we want to abstract over and which we 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, like rebasing the building blocks on top of a realigned foundation.

The new foundation is event orientation, and in order to compose our large-scale applications on top of it we need suitable tools to work with. The

patterns discussed in the second part of this book are a combination of well-worn and comfortable instruments like the Circuit Breaker as well as emerging patterns learnt from wider usage of the Actor model[24]. But a pattern does not only consist of a description of a prototypical solution, more importantly it is characterized by the problem it tries to solve. The main contribution of this book is therefore to discuss the patterns in light of the four tenets of the Reactive Manifesto.

Footnote 24 Originally described by Hewitt, Bishop and Steiger in 1973; also covered among the tools of the trade in chapter two.

1.6.3 Bringing Programming Models Closer to the Real World

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

These design constraints are very familiar from the physical world and from our society: we humans also collaborate on larger tasks, we also perform our 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 systems down in ways which are compatible with the nature of distributed and autonomous implementation.

Of course one should stay away from abuses of anthropomorphisms: we are slowly eliminating terminology like “master / slave” in recognition that not everybody takes the technical context into account when interpreting them[25]. But even responsible use offers plentiful opportunity for spicing up possibly dull work a little, for example by calling a component which is responsible for writing logs to disk a `Scribe`. Then going about implementing that class will have the feel of creating a little robot which will perform certain things that 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 we can turn this around and realize: it’s fun!

Footnote 25 Although terminology offers many interesting side notes, e.g. a “client” is someone who obeys (from latin cluere) while a “server” derives from slave (from latin servus)—so a client–server relationship is somewhat strange when interpreted literally. An example of naming which can easily prompt out-of-context interpretation is a hypothetical method name like harvest_dead_children(); in the interest of reducing non-technical arguments about code it is best to avoid such terms.

1.7 Summary

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

- responsiveness
- resilience
- scalability
- event orientation

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 we have seen the common theme of event orientation as an enabler for meeting the other three challenges.

In the next chapter we introduce the tools of the trade: event loops, futures & promises, reactive extensions and the Actor model. All of these make use of the functional programming paradigm, which we will look at first.