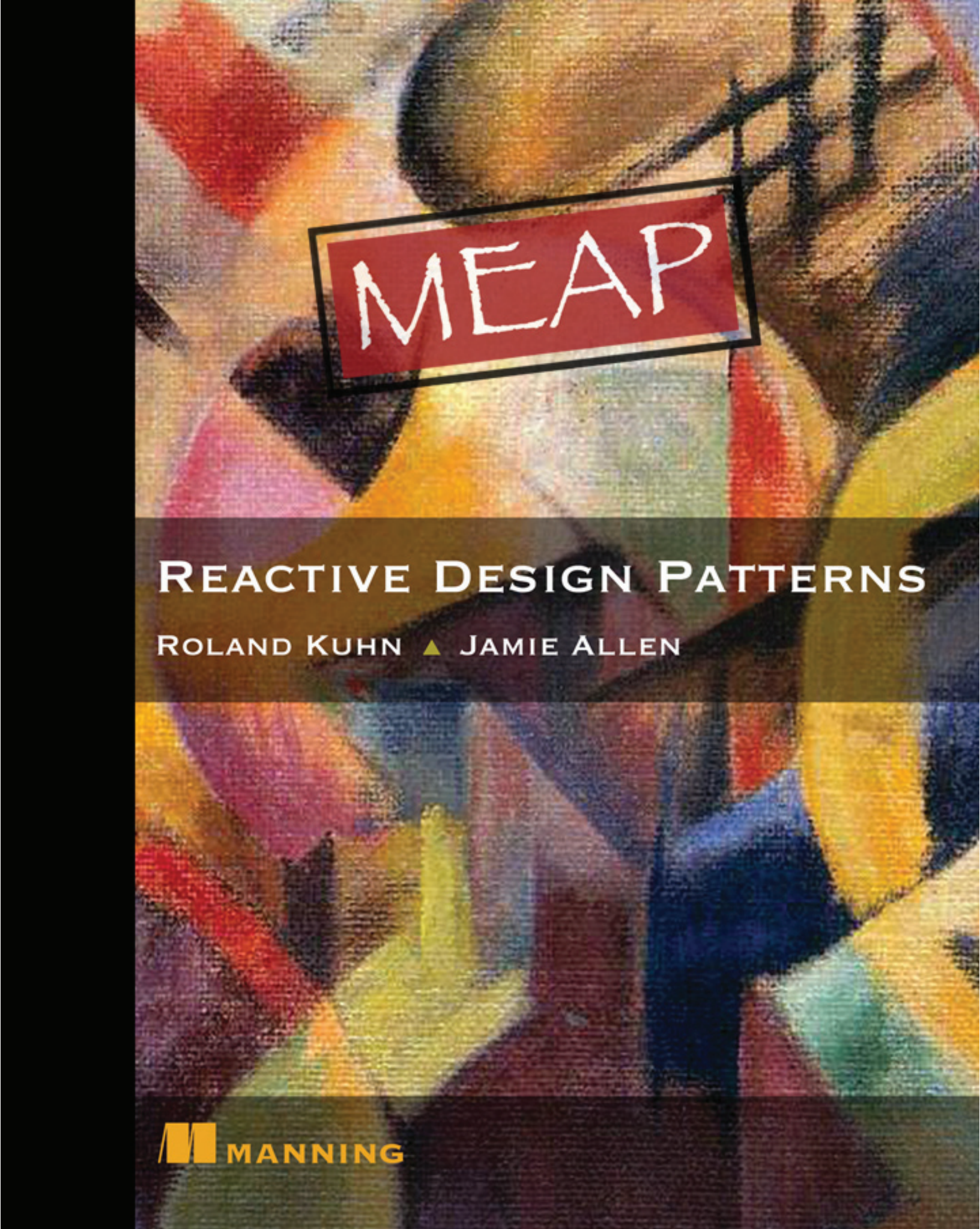
MEAP
REACTIVE DESIGN PATTERNS
ROLAND KUHN ▲ JAMIE ALLEN
MANNING

**MEAP Edition**
**Manning Early Access Program**
# Reactive Design Patterns
**Version 8**



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

# *Welcome*

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 have so far released the first two parts of the book. The first part introduces reactive system design using a concrete example, discusses the reactive manifesto in depth and presents the tools of the trade. The second part focuses on the theory of how the design goals of reactive systems can be implemented: it explains how to structure a system, how to decompose it into its runtime components and how these interact.

Looking ahead to the third part, which we will be working on next, we will be drilling down into the details of how to leverage each of the reactive principles in practice. We cover patterns for testing, fault tolerance, message flow, flow control, persistence, and patterns specific to actors.

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.

Dr. Roland Kuhn

Jamie Allen

# brief contents

# 1 Introduction

Have you ever been wondering how those high-profile web applications are implemented? Those social networks or huge retail sites must certainly have some secret ingredient that makes them work fast and reliably, but what is it? In this book you will learn about the design principles and patterns behind such systems that never fail and are capable of serving the needs of billions of people. While the systems you build may rarely have exactly these ambitious requirements, the primary qualities are very common:

- We want the application to work reliably, even though parts (hardware or software) may fail.
- We want it to keep working once we have more users to support, we want to be able to add or remove resources in order to adapt its capacity to changing demand—capacity planning is hard to get right without a crystal ball.

In the first chapter we sketch the development of an application that exhibits these qualities and some more. We illustrate the challenges that you will encounter and present solutions based on a concrete example—an app that allows users to share their current location and track others on a map—but we do so in a technology-agnostic fashion.

This use-case sets the stage for the detailed discussion of the Reactive Manifesto that follows in chapter two. The manifesto itself is written in the most concise and high-level form in order to concentrate on the essence, namely the combination of individually useful program characteristics into a cohesive whole that is larger than the sum of its parts. We show this by breaking apart the high-level traits into smaller pieces and explaining how everything fits back together.

The introduction is then completed with a whirlwind tour through the tools of the trade: functional programming, Futures and Promises, communicating sequential processes (CSP), observers and observables (reactive extensions), the Actor Model.

# 1 An Illustrated Example

In this chapter we will build an application based on the reactive principles. We do this in order to demonstrate in concrete terms what this means and to help you get into the reactive mind-set for the following chapters.

The application we are building is all about location and movement: users can share their location information with others in real-time and they can communicate using short messages with everyone who is in their vicinity. The location of each individual is hereby used for two purposes.

1. Each user can share their location with a set of other users who will be able to track it on a map.
2. Each user can share their location in anonymized form so that aggregated data can be displayed to all users (like “37 users per hour moving westward on highway 50 through Dodge City, Kansas”).

## 1.1 Geographic Partitioning

How do we construct such an application? One thing is quite clear, most of the information processing will be “local” as in pertaining to some specific place on earth. Therefore we divide the earth up into regions, starting perhaps with one region per continent (plus some for the oceans). The granularity of states within each continent varies greatly, so for simplicity we continue by just cutting each continent along lines of latitude and longitude into 16 tiles, four by four, as shown in figure 1.1:

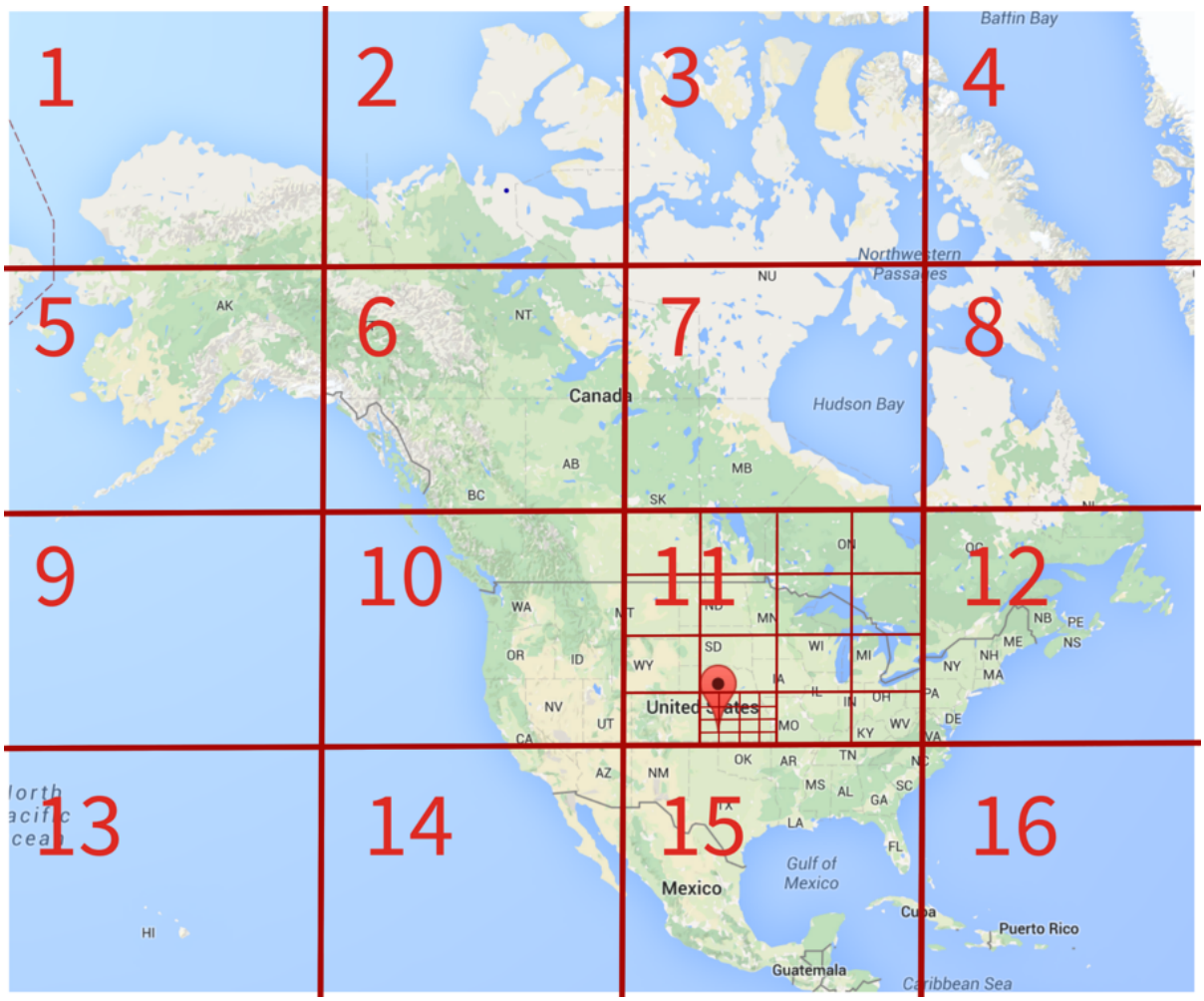


**Figure 1.1 The North American continent, divided recursively into 4 by 4 tiles.**

We continue this process recursively as shown in the picture, quadrupling the resolution in longitude and latitude at each step until it is fine enough—say below one mile in either direction[1]. Now we have a way of associating every possible location on planet earth with a map region, which for Dodge City, Kansas starts with

Footnote 1 There are more refined ways of partitioning a map, but this is a sufficiently simple one that allows us to concentrate on the essence of the program; for further study please refer to R-trees or other literature.

- it is in North America
- … in tile number 11 (at level 1)
- … within that in sub-tile number 14 (at level 2)
- … within that in sub-tile number 9 (at level 3)
- … and so on

Now, when someone shares their location this means that the lowest level map tile containing that location must be informed that there is a user positioned within it. Other users looking at that same map tile can register for (anonymized) updates on what happens within that little map area. The influx of position updates for each of these

lowest level map tiles is given by how many users are logged into the application within that geographic area, and the outflux conversely is given by how many users are watching that precise map tile. This means that no matter how many users may eventually be using this application, we can regulate the amount of information processing per map tile by choosing the granularity—which means choosing the number of levels we go down to in our partitioning.

The first important point about implementing a reactive application is thus to identify the minimal unit of processing that can operate independently. We can execute the book-keeping functions of where people are moving within a map tile separately from the book-keeping of all other map tiles, possibly on different computers or even in different data centers—each catering to one continent for example. This means that we can adapt the processing capacity of the overall system to the actual load by growing or shrinking the number of these processing units: merging two or more into one is not a problem since they were independent to begin with, so the only limit is given by how fine at most we can make this split. The result is a system that can elastically be scaled up and down and thereby react to varying load.

But we are getting ahead of ourselves, since the current design is not yet complete. Users will always—knowingly or not—push our applications to their limits, and in this case there is a rather simple exploit that will ruin our calculations: just zoom out on the map that you are watching, and your interest will cover a large number of map tiles, resulting in a correspondingly large rate of position updates that are requested and sent. For one this will overwhelm the curious user's client with too much data, and for another if too many users do this then our expectations about the outflux of data from a map tile will be exceeded. Both of these mean that more communication bandwidth will be used than we planned for and the consequence will be system overload and failure.

## 1.2 Planning the Flow of Information

In a reactive application each part—each independent unit of processing—reacts to the information it receives. Therefore it is very important to consider which information flows where and how large each of these flows are. The principle data flows for our application are shown in figure 1.2.

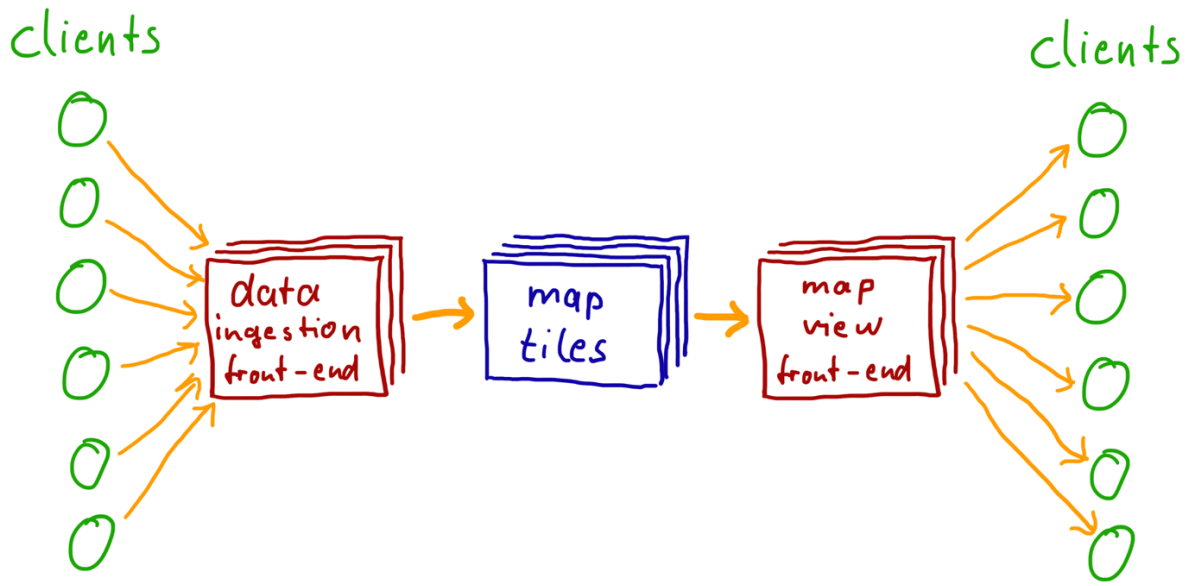


**Figure 1.2 Data are flowing from the clients that submit their position updates through a front-end that handles the client connections into the map tiles and onwards to those clients that are watching the map.**

In our example application we could define that each user sends one position update every five seconds while the app is running on their mobile device—phone, tablet, or wrist watch. We can ascertain that by writing the client app ourselves or by enforcing this limit in the API that our application offers to the author of client apps. Each position update will amount to roughly a hundred bytes, give or take (10 bytes each for timestamp, latitude, and longitude; 40 bytes lower-level protocol overhead for example for TCP/IP; plus some additional space for encryption, authentication, and integrity data). Factoring in some overhead for congestion avoidance and message scheduling between multiple clients, we shall assume that each client's position update streams costs roughly 50 bytes per second on average.

### *1.2.1 First Step: Accepting the Data*

The position updates need to be sent via the internet to a publicly addressable and accessible endpoint; we call this the "front-end node". The current lingua franca for such purposes is HTTP, in which case we need to offer a web service that clients contact in order to transmit their data; the choice of protocol may vary in the future, but the fact remains that we need to plan the capacity of this data ingestion endpoint according to the number of users we expect. The functionality in terms of processing that the endpoint provides is merely to validate the incoming data according to protocol definitions, authenticate clients and verify the integrity of their submitted data; it does not care about the details of the position itself, for those purposes it will forward the sanitized data to the map tile it belongs to.

Common networks today operate at 100–1000 Mbit/sec, we conservatively assume an

available bandwidth of 50 MB/sec half of which we allocate to the reception of position updates and therefore arrive at a capacity of 500,000 clients that can be handled by one front-end node. For the sake of simplicity we also assume that this node's computing resources are sufficient to handle the validation of the corresponding rate of data packets that need to validated, authenticated and verified—otherwise we would reduce the nominal capacity per node accordingly.

With these numbers it is clear that one node will probably suffice for the initial deployment from a data rate perspective; we will want to have two of them in any case for fault tolerance. Serving the whole world population of 7 billion people would hypothetically require 14,000 active network nodes for the data ingestion, then preferably distributed among data centers that are spread across the whole planet and with a healthy percentage of spares for redundancy. The important point to note here is that each of these nodes operates fully independently of all the others, there is communication or coordination necessary between the in order to unwrap, check and route the position updates, which enables us to do this simple back of the envelope estimation of what we would need in order to grow this part of the system to a given scale.

### 1.2.2 Second Step: Getting the Data to their Geographical Home

The function of the front-end node was to accept and sanitize the incoming data and then to send it onwards to the map tile it belongs to. Our rough estimate of the data rates likely applies to the sanitized data as well, we will just have traded the data integrity and authentication data for client IDs and associated data: those were implicit to the client's network connection with the front-end node but now they need to be explicitly incorporated in the data packet on the network connection between the front-end node and each of the map tiles it receives updates for.

Hosting a map tile on a single network node then translates to the ability of handling 500,000 clients within that map area. This means that the tiles need to be small enough so that this limit is never violated. If all map tiles are of the same size—i.e. the same level of partitioning is used throughout the whole map—then there will be tiles that are a lot more frequented than others. Densely populated areas like Manhattan or San Francisco or Tokyo will be close to the limit whereas most of the tiles covering the Pacific Ocean will rarely ever see anybody move on them. We can account for this asymmetry by collocating a number of low-rate map tiles on the same processing node while keeping high-rate tiles on their own nodes.

As we remember it was crucial that the front-end nodes can perform their work independently from each other in order to be able to adjust the system capacity by adding or removing nodes; we will see another reason for this when discussing how we react to failures within the system. But how can we get them to agree on which map tile is hosted by which internal network node? The answer is that we make the routing process simple

and deterministic by having a map tile allocation service disseminate a data structure that describes the placement of all tiles. This data structure can be optimized and compressed by utilizing the hierarchical structure of the map partitioning. Another consideration is that once this application has grown beyond a single data center we can route clients to the right data center that hosts the geographic region they are currently located in, at which point each front-end node only needs to know the location of the tiles its data center is responsible for.

One interesting question at this point is how we react to changes in our application deployment: when a node dies or is manually replaced, or when map tiles are reshuffled in order to adapt to changed user habits, how is this communicated to the front-end nodes? And what happens to updates that are sent to the "wrong" node? The straight-forward answer is that during such a change there will be a time window where position updates pertaining to certain map tiles will simply be lost. As long as we can reasonably expect this outage to be only temporary and of the order of a few seconds long then chances are that nobody will actually notice; one or two missing location updates do not have major effects on the aggregated statistics of a map tile (or can be compensated for) and not seeing a friend's position move on a map for a few seconds once in a blue moon is unlikely to be of consequence.

### *1.2.3 Step Three: Relocating the Data for Efficient Querying*

We have now ensured that the influx of each map tile does not exceed a certain threshold that is given by the capabilities of the processing hardware. The issue that sparked this foray into data rate planning was that the outflux was not limited since clients can zoom out and thereby request and consume more data than they produce and submit.

When we visualize the map that shall show us the movement of all the anonymized users within its area, what do we expect to see when zooming out? We can certainly not follow each individual and track its course once there are more than a few handful of them in the region we are looking at. And when zooming out to view all of Europe, the best we can hope for is aggregate information about population density or average velocity, we will not be able to discern individual positions anyway.

In the same way that we designed the information flow for data ingestion we can take a look at the data extraction as well. Looking at a map of Europe is the easy case because that will not require much data: the large-scale averages and aggregate numbers do not change very quickly. The largest data demand will be given by users that are being tracked individually while being sufficiently closely zoomed in. Let us assume that we allow up to 30 individuals before switching to an aggregated view, and let us further assume that we can limit the data consumption of the aggregate view to be equivalent to those 30 tracked points. One update will have to contain a timestamp and up to 30 tuples of identifier, latitude, and longitude. These can presumably be compressed due to being

within a small map region, perhaps amounting to 15 bytes for each 3-tuple. Including some overall status information we arrive at roughly 500 bytes for a single update, which leaves us with about 100 bytes per second on average for one update every five seconds.

Calculating again with an available network bandwidth of 50MB/sec where half is allocated to the client-facing traffic, this yields a capacity of serving 200,000 map views from a single front-end node (subtracting 20% overhead[2]). These front-end nodes are also answering requests from clients, but they are of a different kind than the ones that are responsible for data ingestion. When a user logs into the app, the mobile device will start sending position updates to the latter ones while every time the user changes the map view on their device a request will be sent to the former ones, registering for the updates to be displayed on the screen. This naturally decouples the two activities and allows a user to view a far-away map region without additional headaches for us implementers.

Footnote 2 Sending from one host to a multitude of others requires less overhead than having a multitude of clients send to a single host, see also the TCP incast problem.

At this point the big question is where these map view front-end nodes get their data from? We have so far only brought the position updates to the lowest-level map tiles and requesting their updates in order to calculate aggregate values will not work: serving 200,000 views could mean having to listen to the updates of millions of map tiles, corresponding to hundreds of terabytes per second.

There is only one solution to this dilemma, we must filter and preprocess the data at their source. Each lowest-level map tile knows the precise location and movement of all users within its geographic region and it would be easy enough to calculate their number, average movement speed and direction, center of gravity, and other interesting quantities. These summary data are then sent every five seconds to the map tile one level up that contains this tile.

As a concrete example consider that the lowest level is 7 partition steps below the North American continent. The center of Dodge City, Kansas on level 7 calculates the summary information and sends it to the encompassing level 6 map tile, which will also receive such summaries from the 15 other level 7 neighbors that it contains. The good thing about aggregate quantities such as user count, center of gravity, and so on is that they can be merged with one another to aggregate at higher and higher granularity (summing up the users, calculating the weighted center of gravity of the individual centers, and so on). The level 6 map tile performs this aggregation every five seconds and sends its summary up to its encompassing level 5 parent, and this process is repeated all the way up to the top level.

The data rate needed for this transfer is fixed to the size of the summary data packet once every five seconds for each sender, and sixteen times that amount for each recipient;

we can assume that each data packet should fit within 100 bytes. In many cases these data do not even need to travel across the network since sparsely populated map areas are collocated on the same processing node and the summary levels can be collocated with the lowest level map tiles as well.

When a map view front-end node now needs to access the summary information at level 4 for displaying a map spanning approximately 100 by 100 miles it will just request the summary information from the roughly sixteen level 4 map tiles covering the viewport. Knowing that network bandwidth will likely be the most limiting factor for these view front-end nodes, it would be possible to optimize their usage on the internal side by redirecting external clients between them such that one front-end node handles many similar requests—for the same approximate geographic region at the same summary level. That way this node can satisfy multiple clients from the same internal data stream. This is shown in figure 1.3.

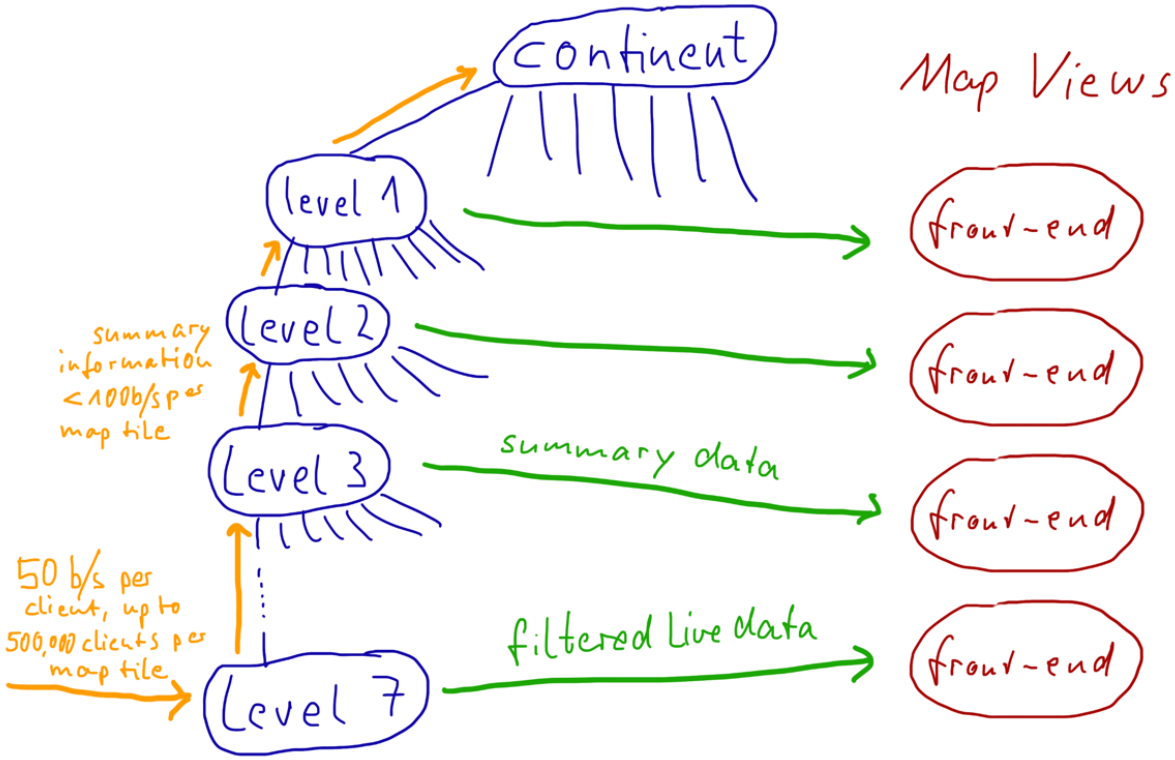


**Figure 1.3 The flow of data from the data ingestion on the left to the map views on the right, with summary data traveling upwards the hierarchical map tile structure.**

The one piece that still needs consideration is how to handle the fully zoomed-in case: when a user points their map at Big Ben to see all the tourists milling about in the center of London then care must be taken to not send all the data from that highly frequented map tile to the front-end node because that could potentially take up all the available bandwidth by itself. We said earlier that a map should display only summary information as soon as the number of individual data points exceeds 30. The calculation of this

summary must happen on the network node that hosts the Big Ben map tile, so the request from the front-end node will contain the coordinates of the desired viewport and then the map tile can determine whether to calculate aggregate information or send along the updates of up to 30 individual user positions, depending on how many people are actually moving within the map area in question.

One aspect of this flow diagram deserves mention, namely that it takes a little while for each new piece of information to make its way up to the top, in this example with seven levels it takes on average about 18 seconds (7 times an average delay of 2.5 seconds). This should not be a problem, though, since the summary information changes much more slowly the higher up we get in the hierarchy.

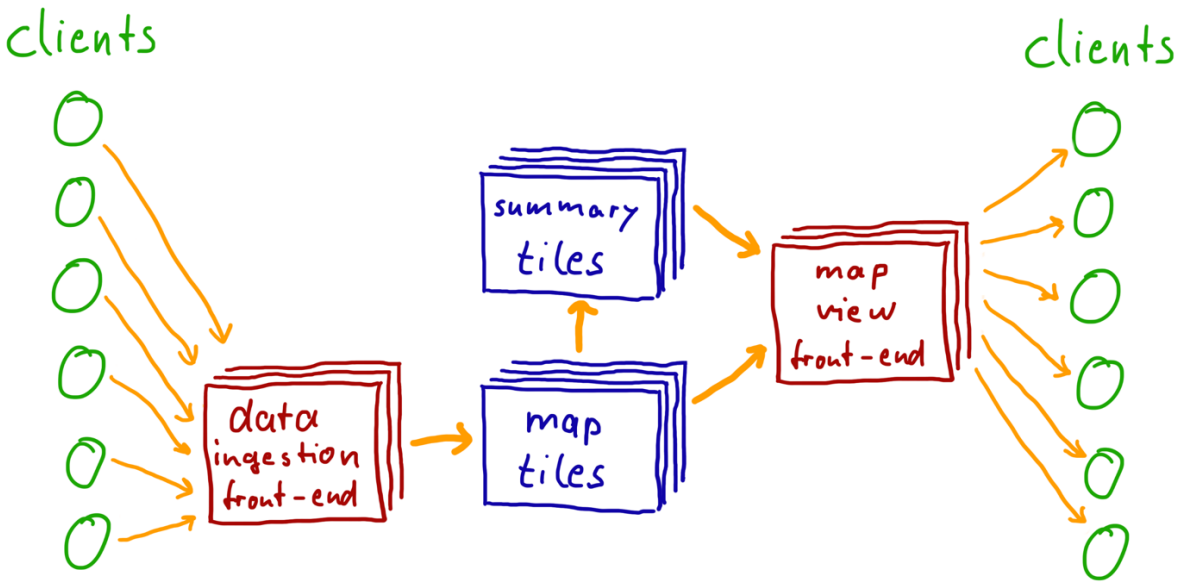


**Figure 1.4 The flows of position updates and summary information through our application, from the position updates generated on the left through the map tile hierarchy towards the map views on the right.**

### *1.2.4 Taking Stock*

What have we achieved so far? We have designed the flow of information through our application as shown in figure 1.4. We have avoided the introduction of a single bottleneck through which the data must pass, all parts of the design can be scaled individually. The front-ends for data ingestion and map views can be adapted to user activity while the map data itself is modeled as a hierarchy of map tiles where the granularity can be chosen by picking the number of partition steps. The processing of the data passing through the map tiles can be deployed onto a number of network nodes as needed in terms of processing and network resources. In the simplest scenario everything can run on a single computer, and at the same time the design supports deployment in a dozen data centers and on thousands of nodes.

## 1.3 What if something fails?

Now that we have a pretty good overview of the parts of our application and of the data flows within it, we should consider how failures will affect it. This is not a black art, on the contrary there is a very simple procedure that we can follow: we consider every node in our processing network and every data flow link one by one and determine what happens if it fails. In order to do this we need a failure model, where a good starting point for a network-based system is the following:

- network links can drop arbitrary bursts of messages (which includes the case of "the link was down for three hours")
- processing nodes can stop responding and never recover (for example by way of a hardware failure)
- processing nodes can intermittently fail to respond (for example due to temporary overload)
- processing nodes can experience arbitrary delays (for example due to garbage collection pauses like for the JVM)

There are more possibilities of what can go wrong and you will need to assess your system requirements carefully in order to decide what else to include; some more choices are that network links may corrupt data packets, data packets may experience arbitrary delay, processing nodes may respond with erroneous data, or they may execute malicious code and perform arbitrary actions. We will also need to consider the effect of executing multiple parts of the application on the same hardware, since this means that hardware failure or resource exhaustion can affect all these parts simultaneously. The more vital the function of your application is for your organization, the more detailed the considered failure model will be. For this example case we stick to the simple list of bullet points outlined above.

### 1.3.1 A Client Fails

Mobile devices can fail for a whole host of reasons, ranging from their destruction over empty batteries to a software crash. Users are used to dealing with those (replacing the phone, charging it or restarting the phone or just the app) and they do not expect things to work while their device has failed. This means that we only need to concern ourselves with the effects of a failure on the internal processes of our application.

Firstly, the stream of position updates will cease. When this happens we might want to generate a visible representation for others who were seeing this user on their map, perhaps changing the color of the marker or making it translucent. The lowest-level map tile will be responsible for tracking the liveness of the users that move within it.

Secondly, the map view for the client will not be able to send updates any longer, network buffers will run full and socket writes will eventually time out. Therefore we must protect the map view front-ends from becoming clogged with defunct client

registrations. This is commonly done by including a heartbeat signal in the protocol and closing the connection when those stop coming in.

### 1.3.2 A Client Network Link Fails

From the perspective of the application it does not matter why position updates cease, the failure of the mobile device or its software is indistinguishable from a failed network connection. The consequences will thus be the same as discussed in the previous section.

From the perspective of the client on the other hand it is in general not distinguishable whether the front-end node it was connected to failed or the network link is at fault, both will look largely the same. Hence the remedy will also be the same as discussed in the next section.

### 1.3.3 A Data Ingestion Front-End Node Fails

The role of such a node is to sanitize and forward the position updates, a failure therefore means that the client will eventually run into trouble while continuing to send data to it. In the same way that the map view monitors the health of its client using heartbeats we can also solve this situation: the client will just reconnect to a different front-end node in case something goes amiss—be that failure temporary or fatal. This is typically realized by placing a network load balancer in front of the pool of real web service endpoint nodes, a strategy that is only possible because it does not matter through which exact node a client sends its updates into the system, the gateways are all equally suited.

The other action that must be taken upon the failure of a front-end node is to properly dispose of it (stopping the application, taking down the machine) and spin up a new instance that starts from a known good configuration. In this way the precise kind of failure is irrelevant, the overall system returns to a fully fault-tolerant state by doing the most robust and simple thing possible. This also means that whatever went wrong is contained within the removed node and cannot spread to the others. The recovery itself must be initiated by a separate service, one that cannot be infected with the failure; we call this a supervisor service. The supervisor monitors its subordinates for proper function and as described takes corrective action in cases of need.

### 1.3.4 A Network Link from Data Ingestion to Map Tile Fails

This situation has no negative impact on the overall function of the application itself, its effect is the same as if the connected clients stop sending position updates to the affected map tile. Therefore depending on which communication protocol is used for this network link both parties should monitor the health of their connection and release all associated resources if it becomes stale.

The simplicity of this problem and of its solution is due to the fact that neither side—front-end node or map tile—depends on the other for correct function. Data flow only in one direction from one to the other, and if data stop flowing then both sides know

how to deal with that. This is what we call *loose coupling* and it is essential for achieving robust failure handling.

### 1.3.5 A Map Tile Processing Node Fails

Since this is the heart of our application we should consider the different failure modes more carefully.

- *Hardware failure:* In case of a node crash all map tiles that were hosted by this node will be failed with it as a result. The front-end nodes will eventually notice this and stop sending updates, but what we need to do is to recover from this situation. The front-ends cannot be responsible for that since it would involve coordination as to who performs the necessary steps, therefore we install a supervisor service that monitors all map tile nodes and spins up a new instance in case of a crash. We discussed earlier that this service will then update the routing knowledge of all front-end nodes so that they start sending updates to the new destination.
- *Temporary overload:* If a map tile sees more traffic than was planned for then it will need to be moved to a separate processing node, otherwise it will take away resources from all its collocated neighbors and the overload will spread and turn into a node failure. This scenario must also be handled by the supervisor service, which for this purpose will need to gather usage statistics and if necessary rebalance the map tiles across the available nodes. If the load is increasing in all parts of the system then this supervisor should also be able to request new nodes to be brought online so that the additional load can be handled. Conversely, once load drops significantly the supervisor should reallocate map tiles to free up and release redundant nodes.
- *Permanent overload:* It is also possible that the partitioning of our map is not done adequately and a single map tile is hit consistently by too many requests. Since we cannot split or reallocate this one, such a failure will need to raise an alert when it is detected and the system configuration must be adapted manually to correct the mistake.
- *Processing delays:* In some cases the inability of processing new data lasts only for a few seconds while e.g. the JVM is performing a major garbage collection cycle. In such cases no specific recovery mechanism is necessary beyond possibly dropping some updates that are outdated by the time the machine comes back to life. There is a point, of course, where such an interruption is mistaken for a node crash, we will have to configure the supervisor service to tolerate pauses up to a given duration and take corrective measures once that is exceeded.

As in the case of a front-end node failure we encounter the need for a supervisor service, one that keeps an eye on the all the deployed processing nodes and can heal the system in case of failure—by making use of its global view and if necessary by disposing of faulty instances and creating fresh ones. The supervisor does not become a bottleneck in the system because we keep it outside of the main information flows of our application.

### 1.3.6 A Summary Map Tile Fails

These processing units are very similar in function to the lowest-level map tiles, and in fact they are part of the same information routing infrastructure, hence we supervise them in the same fashion.

### 1.3.7 A Network Link between Map Tiles Fails

This case is very similar to the front-end nodes being unable to forward position updates to the map tiles—data will just not arrive while the failure lasts. We need network monitoring place so that the operations crew is notified and will fix the issue, and apart from that we will just have to throw away data as they grow stale. This last part is very important to avoid the so-called *thundering herd problem* when network connectivity is restored: if all data are buffered and then sent at once, the destination will likely be overloaded as a result. Luckily we do not need to buffer data for long periods of time within this part of our application, since all we are modeling is a live view on a map with no historic information, lost updates are just a fact of life.

### 1.3.8 A Map View Front-End Node Fails

In this case we can act in the same fashion as for the data ingestion front-end nodes in that we have clients reconnect through a load balancer as soon as they determine that something is wrong, and we have a supervisor service that will dispose of nodes and provision new ones when needed. The latter actions can also occur in response to changes in load, this way the monitoring by the supervisor enables the system to elastically scale up and down as needed.

There is one more consideration in this case that we must take care of, namely that map view updates are sent by the map tiles according to the front-ends' registrations. If a front-end becomes unavailable and is replaced then the map tiles need to stop sending data their way as soon as possible since the new client registrations that replace the failed ones will soon take up their share of the planned bandwidth again. Therefore the map tiles need to closely monitor their map views and drop updates when they are not consumed in a timely fashion.

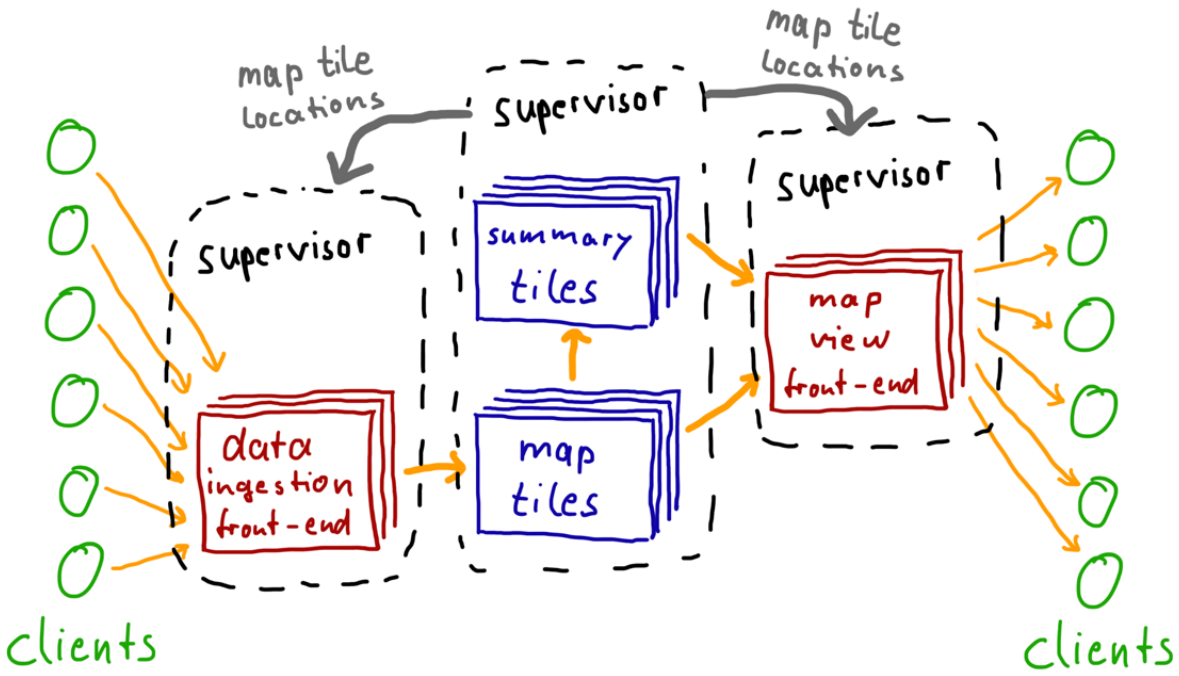


**Figure 1.5 Deployment structure of the application with supervisor services and their relationship: the map tiles supervisor informs the front-end supervisors of where position updates and map view registrations shall go.**

### *1.3.9 Failure Handling Summary*

While we systematically walked along all data flows and considered the consequences of node or communication failures we have encountered two main needs:

- Firstly, communication partners frequently are required to monitor the availability of their interlocutors; where no steady stream of messages is readily available it can be generated using a heartbeat mechanism.
- Secondly, processing nodes must be monitored by supervising services in order to detect failures and take corrective action.

Figure 1.5 shows the complete deployment structure of our application with the added supervisors. It also notes that the service that is supervising the map tiles must inform both types of front-end nodes of the current mapping where each map tile is hosted.

## *1.4 What have we Learnt from this Example?*

We have modeled an application that can serve any number of clients, allowing them to share their location and to see how others are moving on a map. The design was done such that we can easily scale the capacity from trying it out on a development notebook—running all parts locally—to hypothetically supporting the use by all humans on planet earth. The latter will require considerable resources and their operation will still be a large effort, but from the technical side the application is prepared for that. We have achieved that by considering foremost the information that will be processed by the application and the main flows of data that are necessary for this.

The most important characteristic was that data flow always forward, from their source (the position updates of the mobile devices) via processing stages (the map tiles) towards their final destination (the map displayed on the mobile devices). The processing nodes on this path of information are *loosely coupled* in that failures of one are dealt with in the same way as communication outages.

We have built resilience into the design by considering the major parts of the application to be isolated from each other, only communicating over networks. If multiple parts are running on the same machine then a failure of the machine—or a resource exhaustion caused by one part—will make all of them fail simultaneously. It is especially important for achieving fault tolerance that the services that are tasked with repairing the system after failures, the supervisors, are isolated from the other parts and are running on their own resources.

In this way we have experienced all the main tenets of the Reactive Manifesto:

- Our application is *responsive* due to the resource planning and map partitioning we have done.
- Our application is *resilient* since we have built in mechanisms for repairing failed components and for connecting to properly functioning ones instead.
- Our application is *elastic* by monitoring the load experienced by the different parts and by being able to redistribute the load when it changes, a feat that is possible due to having no global bottlenecks and processing units that can work independently from each other.
- All of this is enabled by *message-driven* communication between the parts of our application.

## 1.5 Where do we go from here?

The attentive reader will have noticed that not all functionality has been implemented for our example application. We have detailed how to implement the second requirement of sharing anonymized position updates while we left out the first requirement that a user shall be able to share their location with a set of other users. It will be a good exercise to apply the same reasoning also in this case, designing additional parts to the application that keep data in a user-centric fashion rather the map-centric one we have built so far. Modifications of the trust relationship between users will have to be more reliable than position updates, but they will also be vastly less frequent.

While thinking about the issues you find in this new part of the application you will likely encounter difficulties as to how you can achieve resilience and elasticity and how you define and maintain proper responsiveness for them. That will be a good point at which to delve into the following chapters of this book to learn more about the underlying concepts of reactive application architecture and the philosophy that ties them together. The resulting patterns that emerge when applying reactive design are covered in

the third part of this book and once you have read it you should have no difficulty at all to design the user-centric part of this example application plus any additional functionality you can think of.

# 2 *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[3]: 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 or more generally to its environment. This is the core motivation behind reactive programming and the other reactive traits are derived from this by adding further constraints

Footnote 3 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[4] 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 4 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[5], which was written to define a common vocabulary and to lay out the basic challenges which a modern computer system needs to meet:

Footnote 5 http://reactivemanifesto.org/

- it must react to its users *(responsive)*
- it must react to failure and stay available *(resilient)*
- it must react to variable load conditions *(elastic)*
- it must react to inputs *(message-driven)*

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 inputs without having a way to know beforehand which to expect next.

In the following chapters 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 three. 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.

## 2.1 Systems and their Users

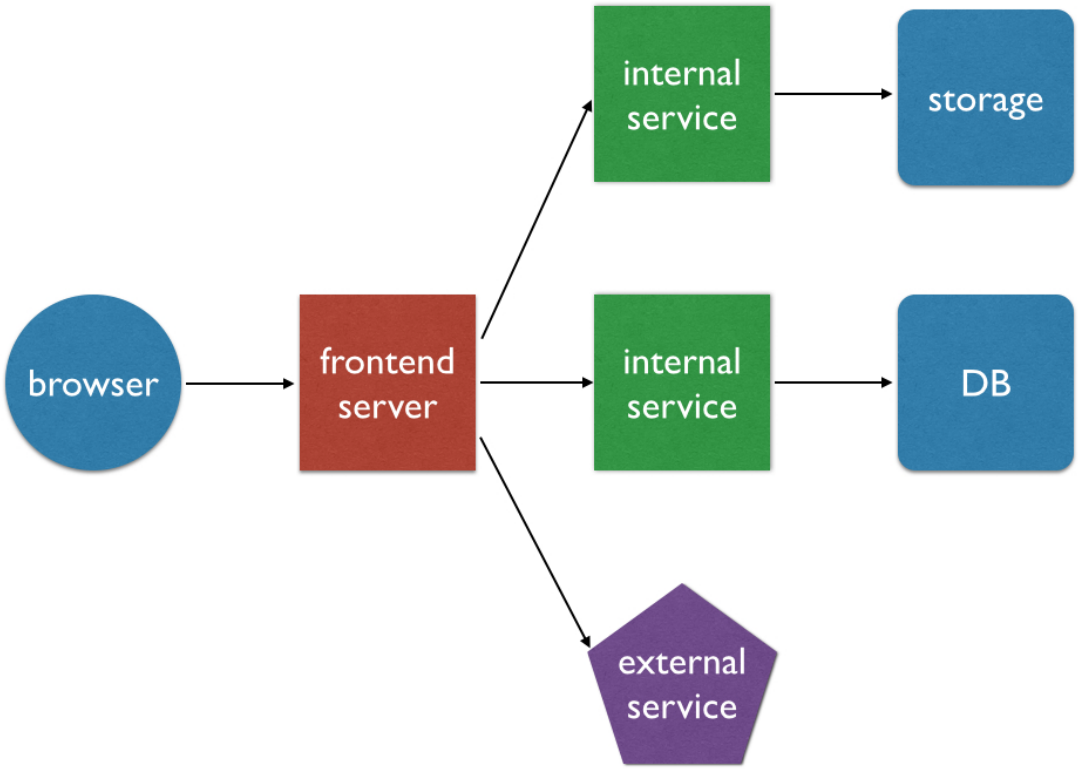


**Figure 2.1 Schematic view of a typical web application deployment with different back-end services: the front-end server dispatches to different internal services, which in turn make use of other services like storage back-ends or databases.**

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 2.1 depicts 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.

## 2.2 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.

### 2.2.1 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 factorial 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[6] 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 6 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.

### 2.2.2 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[7]; 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 7 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*[8]—is known to the user, and that the latency of the service is consistent enough for this purpose.

Footnote 8 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.

### 2.2.3 Cutting Down Latency by Parallelization

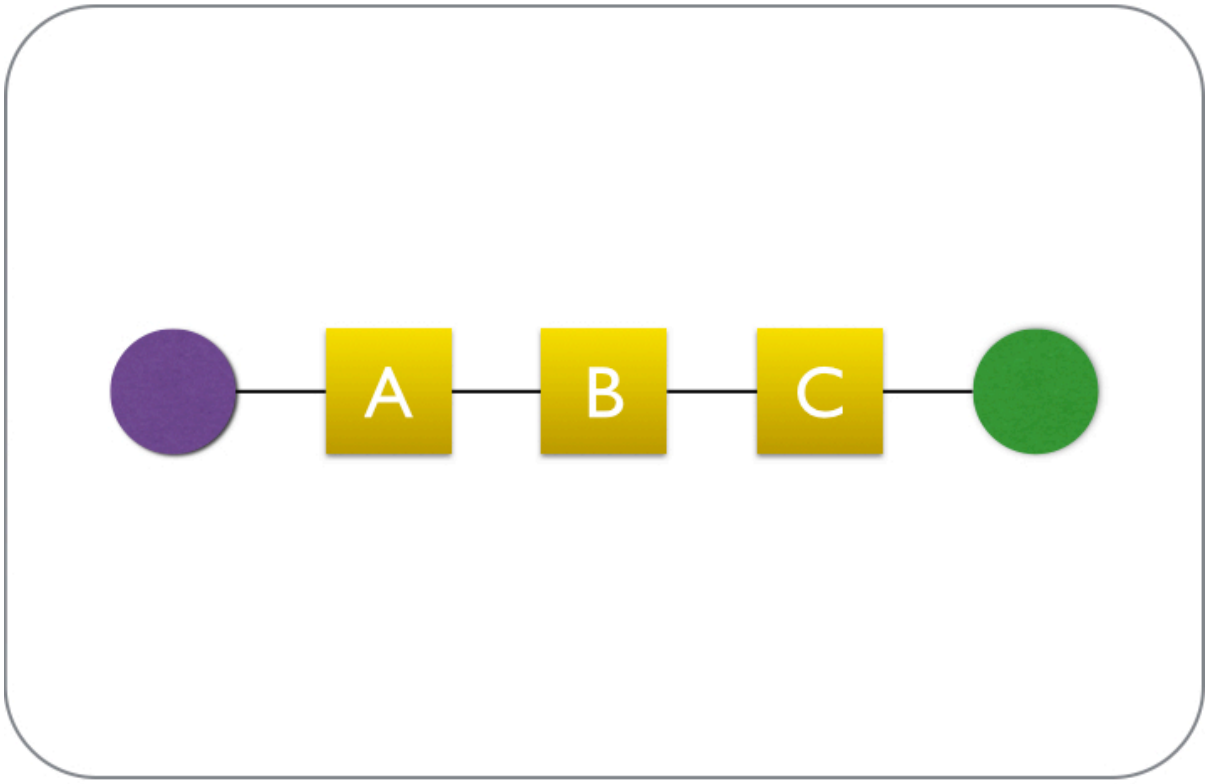


**Figure 2.2 A task consisting of three sub-tasks that are executed sequentially: the total response latency is given by the sum of the three individual latencies.**

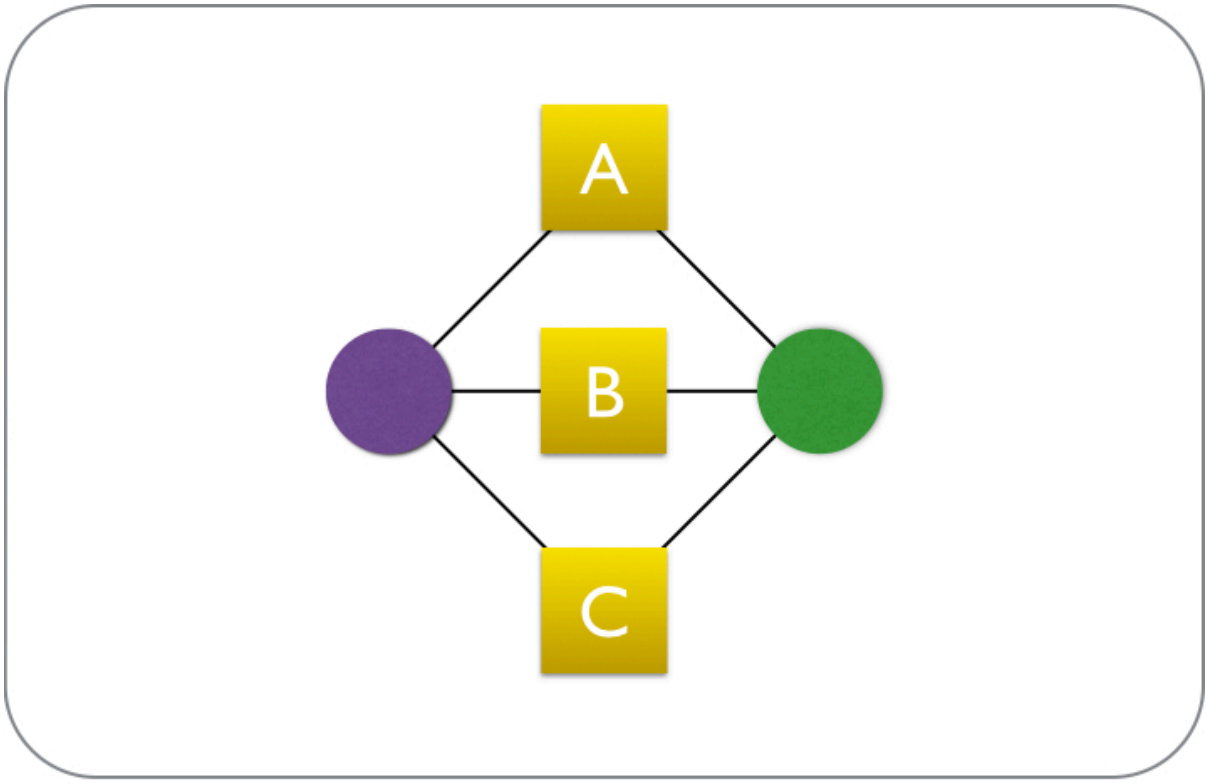


**Figure 2.3 A task consisting of three sub-tasks that are executed in parallel: the total**

**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 2.3 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 2.2 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 *task-oriented* in order to support this. The example from above would transform into the following[9]:

Footnote 9 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 task-oriented 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.

### 2.2.4 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 the 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.

### 2.2.5 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 task-oriented system usually has a mechanism which enqueues tasks 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[10]. 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 10 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 2.4.

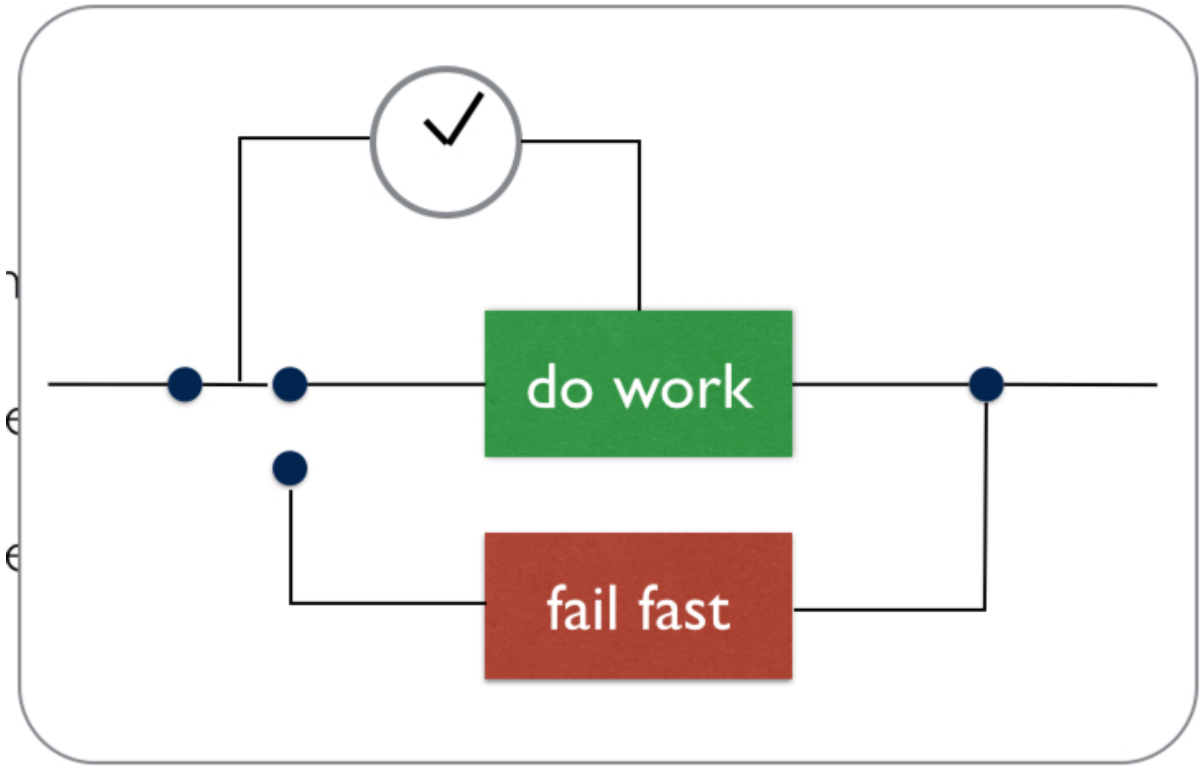


**Figure 2.4 A circuit breaker in electrical engineering protects a circuit from being destroyed by a too high current. The software equivalent does the same thing for a service which would otherwise be overwhelmed by too many requests.**

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.

### 2.2.6 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

## 2.3 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[11] 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 11 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. The same must also apply to those failure cases which were not foreseen explicitly in the design, knowing that these will happen as well.

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

There is only one generic way to protect your system from failing as a whole when a part fails: *distribute* and *compartmentalize*. The former can informally be translated as “don’t put all 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.

### 2.3.1 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*.

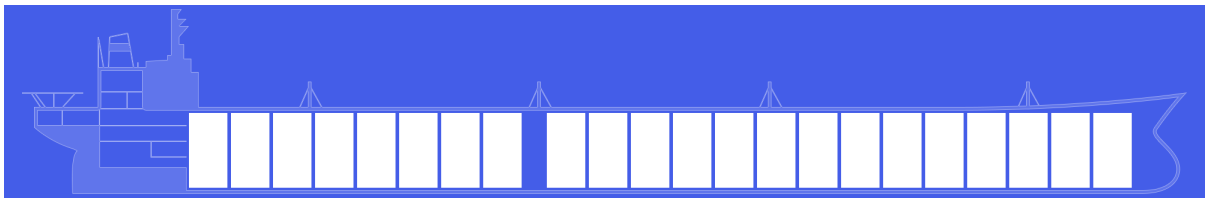

**Figure 2.5 The term “bulkheading” comes from ship building and means that the vessel is segmented into fully isolated compartments.**

Figure 2.5 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[12]. 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 12 "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.

### 2.3.2 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 task-oriented 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[13].

Footnote 13 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 message-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, the request is sent as a message to another execution resource and the response will be a message which needs to be handled when it occurs in the future; 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.

### 2.3.3 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.

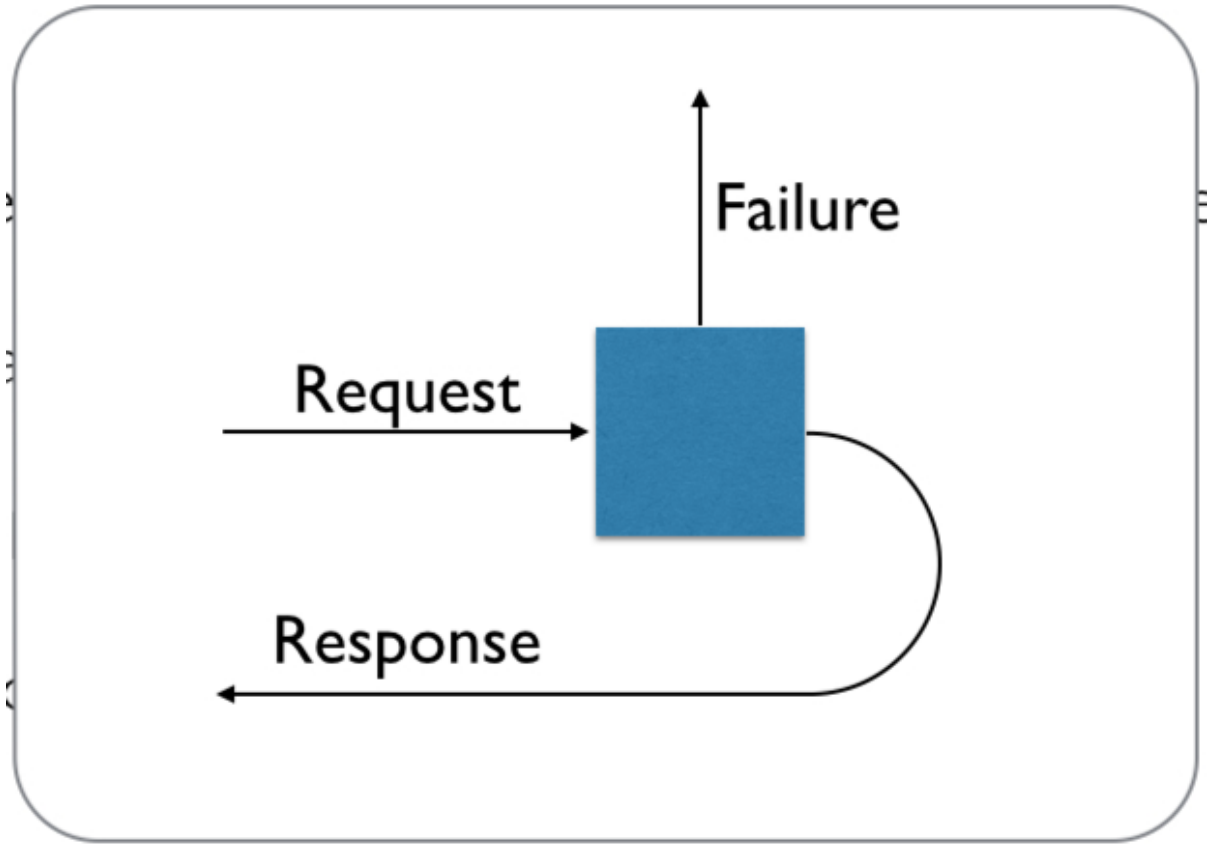


**Figure 2.6 Supervision means that normal requests and responses (including negative ones such as validation errors) flow separately from failures: while the former are exchanged between the user and the service, the latter travel from the service to its supervisor.**

Figure 2.6 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.

### 2.3.4 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.

## 2.4 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[14] 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 14 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 a 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.

### 2.4.1 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 *lambda* at which requests arrive and the average time *W* a request stays in the service while being processed:

*L=lambda•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.

### 2.4.2 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 a message-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

*lambda* = average(requestsPerSecond)
*W* = average(processingTime)
*u* = targetUtilization
*L* = *lambda* • *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 [15] 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 15 You will want to avoid reacting on pure fluctuations in order to keep operations smooth.

### 2.4.3 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 message-driven.
- Change number of active instances based on monitoring data at runtime.

## 2.5 Reacting to Inputs

All of the tenets of the Reactive Manifesto which we have considered so far lead to a task-oriented and message-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 messages and protocols instead of traditional method calls also has benefits on its own.

### 2.5.1 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[16].

Footnote 16 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 messages 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.

### 2.5.2 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 [17] 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 17 The finite speed of light as well as power dissipation make further increases in clock frequency impractical.

### AMDAHL'S LAW

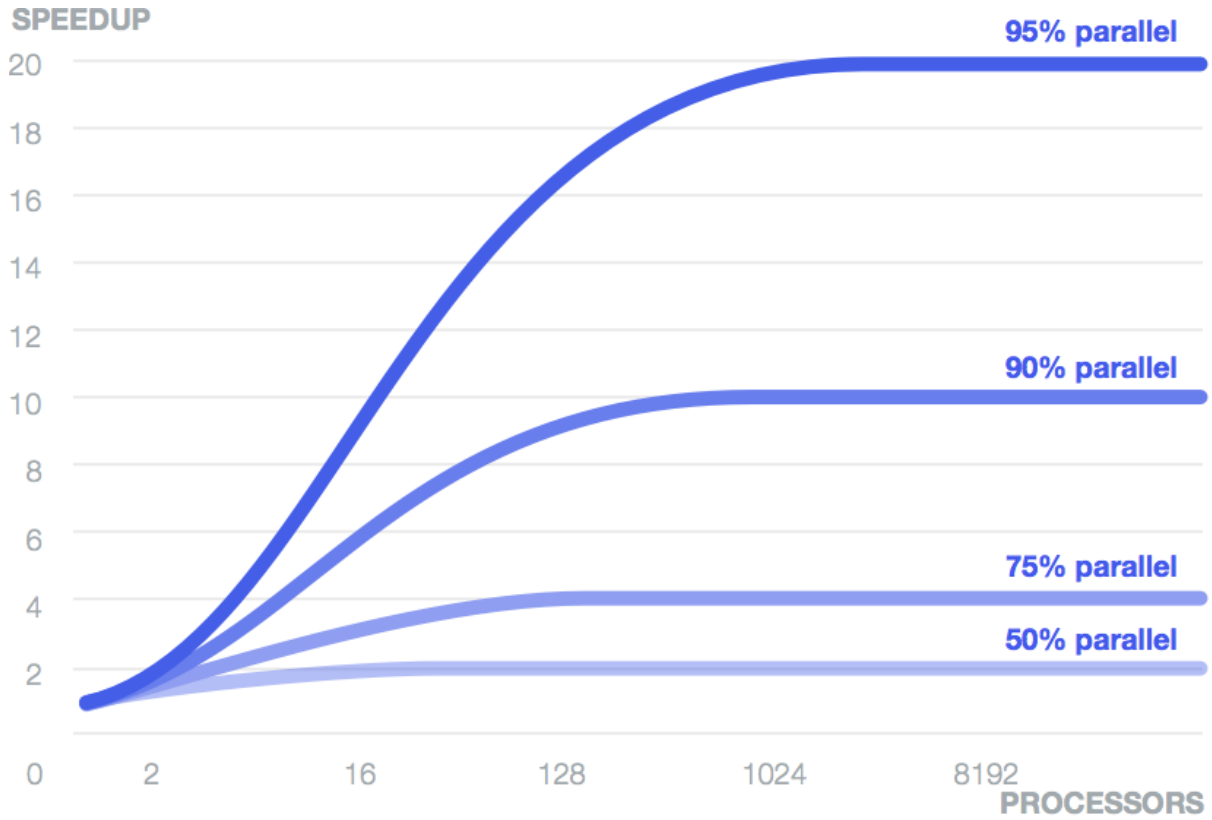


**Figure 2.7 The speed-up of a program using multiple processors in parallel computing is limited by the sequential fraction of the program. For example if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20 times, no matter how many processors 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 messages 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 messages and reacting to messages under the hood, but this fact is completely hidden in order to construct a synchronous façade on top of the underlying message-driven system. 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[18]. 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[19] 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 18 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 19 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 message-driven structure waste CPU resources. If the messages were made explicit in the API such that instead of suspending a thread you would just suspend the computation—freeing up the thread to do something else—then this overhead would be reduced substantially:

```
// 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.

### 2.5.3 Summarizing the Why and How of Event Orientation

The main benefits of message-driven designs are:

- Message 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 a message 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.

## 2.6 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.

### 2.6.1 The Loss of Strong Consistency

One of the most famous theoretical results on distributed systems is Eric Brewer's CAP theorem[20], which states that any networked shared-data system can have at most two of three desirable properties:

Footnote 20 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[21] transaction semantics. For this reason distributed systems have so far concentrated critical components in such a way that provided strong consistency.

Footnote 21 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 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[22] 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 22 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[23]. These have the property of merging cleanly when the partition ends regardless of the modifications that were done on either side.

Footnote 23 Conflict-free Replicated Data Types

Google Docs employ a similar technique called Operational Transformation[24]. 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 24 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 that 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[25]: 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 25 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 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.

### 2.6.2 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 message-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, akin to rebasing the building blocks on top of a realigned foundation.

The new foundation is message 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 third 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[26]. 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 26 Originally described by Hewitt, Bishop and Steiger in 1973; also covered among the tools of the trade in chapter two.

### 2.6.3 Bringing Programming Models Closer to the Real World

The final remark on the consequences of reactive programming takes up the strands which shone 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[27]. 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 27 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.

## 2.7 Summary

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

- responsive
- resilient
- elastic
- message-driven

We have shown how the need to stay responsive in the face of component failure defines resilience, and likewise how the desire to withstand surges in the incoming load elucidates the meaning of scalability. Throughout this discussion we have seen the common theme of message 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.

# 3 Tools of the Trade

The previous chapter explained the reasoning of why we need to be Reactive. It is time to turn our attention to the question of how we can achieve this goal. We want to build applications with both software and hardware infrastructure that will scale in the face of significant load and bursts or spikes of traffic, handling load by degrading gracefully rather than locking up completely. In this manner, the application is able to respond to users at all times. In this chapter, you will learn:

- The impact of choosing specific tools
- The earliest Reactive approaches
- High-level implementations of Reactive solutions

## 3.1 The Impact of Choosing Non-Reactive Tools

Imagine a startup that needs to raise money in the venture capital (VC) world. It can be very easy to build a prototype by using frameworks that support rapid development, and it may also be very easy to find developers skilled in creating applications with such tools. The company builds the prototype and presents it to VC firms in the hope of raising money. If they are successful, they need to go into production, so they quickly turn their prototype application into a production deployment by leveraging a cloud "Platform as a Service" (PaaS)[28] platform. They they begin to market their capabilities so that customers can find them.

Footnote 28 http://en.wikipedia.org/wiki/Platform_as_a_service

However, not all frameworks are created equal from a scalability perspective, and some may have limitations that make it difficult to handle many concurrent connections. There may be some workarounds, but in the end, they may not be enough for an application that will scale as the new company's user base grows. As a result, they are forced to add new virtual machines on their hosting service to handle all of the customer requests that are now flowing in. The scalability might be theoretically linear (where

adding an additional server would increase your capacity by a multiple; for example, 2 servers doubles your capacity, 3 servers triples it, etc). However, this assumes that the hosting provider routes requests to server instances in a fashion that will distribute load to all servers equally, but not all cloud platforms will do that for you. Some providers may distribute load in a random fashion instead of a round robin strategy (where each server receives a request in order before starting at the first server again), which means that merely adding new servers does not mean that they will assist in handling the additional load. As a result, this new company is forced to add even more virtual machines to host the server application than would be necessary if distribution were even, based on the probability of load distribution using the request routing algorithm of the cloud provider.

### *3.1.1 The Cost*

At this point, the new company finds itself running hundreds of virtual machines on the cloud platform. Their theoretical maximum number of concurrent connections that can be handled by this armada of servers is limited to a small multiple of the number of servers due to the limitations of the implementing framework, and may be even less given the variability of the concurrency factors of the tooling and the routing distribution strategy mandated by the cloud provider. The cost of running so many machines is non-trivial—in today's dollars, an individual virtual machine instance on such a "Platform as a Service" hosting provider may cost as much over US$100 per month. For 250 such virtual machines, the cost is now US$25,000/month, at a total cost of US$300,000 per year! That is a very heavy "burn" rate (the rate at which they spend their precious initial investment money) for any startup to withstand.

Such an application might also not be fault tolerant. There may be no way through this platform to coordinate servers without using an external tool, nor does the framework provide constructs that help you manage failure within the logic of the application. So, this new startup is paying too much money while attempting to scale an application in the face of comparatively low traffic while utilizing an architecture that does not have core constructs that support resilience. Worse, what if the language and framework used to implement the service is also very slow at runtime, partly because it uses a dynamic type system (where types of values are not declared in the source code and the runtime must make assumptions about the type of the value when evaluated) that is unlikely to execute as fast as a language with static types that has proven the existence of methods and compatibility at compile time. There are other hidden costs to consider, such as the environmental impact of this approach—you are using more virtual machines to handle a relatively small amount of traffic, which requires more power in the data center.

The startup discussed earlier in this chapter did not choose tooling that allowed them to build a Reactive application, and they pay the costs associated with that decision every single day. Now that we have an understanding of the impact of choosing a specific tool

with which to build our application, we can begin to think about the programming constructs that will be most supportive to building Reactive systems. In this chapter, several technologies will be presented at a high level to describe tools and strategies that can alleviate the costs that this startup now faces. But first, it's important to look back at the history of Reactive solutions.

Early Reactive Solutions

The startup example above is not contrived; there are companies suffering through these very issues right now. For such a company to lower their costs and be more responsive to their users, they need to migrate their application away from the inefficient language and framework to those that will help them do that, much as Twitter did[29] when re-architecting their application away from Ruby on Rails with a more scalable platform. Over the past 30 years, many such tools and paradigms have been defined to do help us build applications that do meet the Reactive paradigm. One of the oldest and most notable is the Erlang language[30], created by Joe Armstrong and his team at Ericsson in the mid-1980s. Erlang was the first language that leveraged Actors (described later in this chapter) to gain mainstream popularity.

Footnote 29 An example of one such effort can be found on the Official Twitter blog here: https://blog.twitter.com/2011/twitter-search-now-3x-faster

Footnote 30 http://www.erlang.org/

Joe and his team faced a daunting challenge - to build a language that would support the creation of applications that would be deployed in a distributed environment and be incredibly resistant to failure. The language evolved in the Ericsson laboratory over time, culminating with the usage of Erlang to build the AXD301 switch in the late 1990s, which reportedly achieved "nine 9s" of uptime. Consider exactly what that means. "Nine 9s" of uptime is the equivalent to saying that an application will be available 99.9999999% of the time. For a single application running on a single machine, there would be roughly 3 seconds of downtime in 100 years!

```
100 years
    * 365 days/year
    * 24 hours/day
    * 60 minutes/day
    * 60 seconds/minute
        = 3,153,600,000 seconds

3,153,600,000 seconds
    * 0.0000001 expected downtime
        = 3.1536 seconds of downtime in 100 years
```

Of course, such uptime of an application running on a single box is purely theoretical;

as of the publication of this book, no application could possibly have been running continuously on a machine longer than modern computers have existed. The actual study upon which this claim was based was performed by British Telecom in 2002 through 2003[31] and involved 14 nodes and a calculation via 5 node-years of study. Such approximations of application downtime depend as much on the hardware as the application itself, as even this most resilient application will not be fault tolerant if it were running on unreliable computers. But such theoretical uptime of the application and the resulting effect on the fault tolerance dimension of Reactive is highly desirable. Amazingly, no other language or platform has made similar claims since the release of this product.

Footnote 31
http://www.erlang-factory.com/upload/presentations/243/ErlangFactorySFBay2010-MatsCronqvist.pdf

At the same time, Erlang was created as a language with dynamic types, and it copies message data for every message it passes between actors. The data has to be copied because there is no shared heap space between two actor processes in the Beam VM. This means that data that will be sent between actors must be copied into a new memory space on the receiving actor process' heap prior to sending the message, to guarantee isolation between the actors and prevent concurrent access to the data being passed.

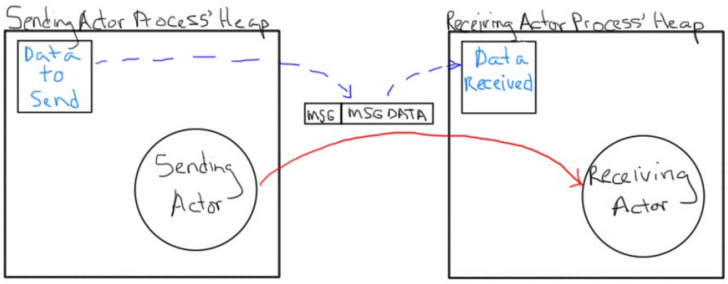


**Figure 3.1 Illustration of data in the sending Erlang actor's heap being transferred via message to the heap of a second Erlang actor who will receive the message**

While these features provide additional safety, where any Erlang actor can receive any message and no data can be shared, they have the effect of lowering the potential

throughput per instance of an application built with it. This means that Erlang deployments have to use more servers than applications built with other tools and platforms in order to handle the same load.

## 3.2 Functional Programming

Functional programming concepts have been around for a very long time, but only recently have they gained favor in the programming language world. But why did Functional Programming disappear from the mainstream for so long, and why is its popularity surging now?

The period between 1995 and 2008 was essentially the "Dark Ages" of functional programming, as the languages such as C, C++ and Java grew in usage and the imperative and Object-Oriented programming style became the most popular way to write applications and solve problems. The advent of multiple cores and their increased availability led to a greater ability for developers to use parallelized implementations, but imperative programming constructs with side effects can be very difficult to reason about in such an environment. Imagine a C or C++ developer who already has the burden of managing their own memory usage in their single-threaded application. In a multi-core world, they now have to do so across multiple threads at the same time, while also trying to figure out who has the ability to access shared mutable memory at what time. This makes something that was already very hard to do and verify in the first place into something that is daunting for even the most senior C/C++ developers.

This has led to a veritable "Renaissance" in functional programming; more languages have designed constructs that follow the functional approach because the paradigm is more supportive of reasoning about problems in concurrent and parallelized applications. By writing code in a functional style, developers have an easier time reasoning about what their application is doing at any given point in time.

The core concepts of functional programming have been around for many years, defined through Lambda Calculus by Alonzo Church[32] in the 1930s. However, the essence of functional programming is the idea that programs can be written using pure mathematical functions (and therefore maintaining *purity* of our functions) - those that return the same value every time when passed the same inputs - that do not incur any "side effects" (see section below). This relationship to mathematics is important - the core ideas of functional programming are that the way you write code is analogous to how functions are composed in math. With all of these powerful tools at our disposal, it is truly a wonderful time to be a programmer again because we can solve our problems with languages that support one or both paradigms simultaneously.

Footnote 32 Alonzo Church, The Calculi of Lambda-Conversion, (Annals of Mathematical Studies Vol. 6), Princeton, Princeton University Press, London, Humphrey Milford and Oxford University Press, 1941.

Functional programming is easy to conceptualize from the mathematical function

perspective, but there are other core concepts to this style that must be considered in order to maintain the purity of our functions.

### 3.2.1 Immutability

In a purely functional program, mutable state is impure and considered dangerous—the same name for a variable can refer to something different at different points in time. Mutable state is any variable that is not stable or final, and can be changed or updated inside of an application. By using final, immutable values in an application, a programmer has a greater ability to reason about what a value will be at a given time, because they know that no one else can possibly have changed it. This is the basis for immutable data structures, where any action performed on the data structure results in a new data structure being created that represents the updated view. The original data structure, however, has remained unchanged, and any other part of the program that continues to use it does not see the change that was made.

By leveraging immutability throughout an application, a developer is able to limit the places where mutation can take place to a very small section of code. In doing so, the possibility of contention, where multiple threads are attempting to access the same resource at the same time and some are forced to wait their turn, is limited in scope to a small region. Contention is one of the biggest prices to pay when trying to make an application by using multiple CPU cores, and should be avoided as much as possible.

### 3.2.2 Side Effects

Side effects are operations inside of a function that could change each time you call it; such as modifying some state (such as a counter of how many times it was called), writing log output to the console, sending data over a network, etc. Utilizing pure functions means that a developer is calling into functions that have no side effects. Some functional programming languages, such as Haskell, enforce rules about where side effects can exist and when they can take place, which is a powerful way for developers to reason about the correctness of their code.

### 3.2.3 Referential Transparency

The concept of referential transparency is that replacing an entire expression with a single value will have no impact on the execution of the program.[33] A more concrete way to think about it is that performing an operation on a particular datum or data set will result in a new value being returned, with no impact on the original data upon which you performed the operation, and no side effects will occur. When using an immutable list, the act of adding, removing or updating a value in that data structure should result in a new list being created with the changed values, and any part of the program still observing the original list sees no change.

Footnote 33 http://en.wikipedia.org/wiki/Referential_transparency_(computer_science)

Consider Java's java.lang.StringBuffer class. If we were to call the reverse method on a StringBuffer instance, we will have a new instance of the StringBuffer with the values reversed. However, the original StringBuffer instance was also changed by making this call.

```
StringBuffer myStringBuffer = new StringBuffer("foo");
StringBuffer myNewStringBuffer = myStringBuffer.reverse();
System.out.println(String.format(
    "myStringBuffer: %s, new value: %s",
    myStringBuffer,
    myNewStringBuffer));

// Result - myStringBuffer: oof, new value: oof
```

This is an example of referential opacity, where the value upon which you performed the operation does change when evaluated, and the expression cannot be replaced by a single value without altering the way a program executes. It's worth noting that the same issues exists for the java.lang.StringBuilder class. In functional programming, referential transparency is required to reason about our application at runtime.

### 3.2.4 Functions as First-Class Citizens

This concept is a bit more abstract than the others, where a function is a value just like an Integer or a String and can be passed into another function or method. The idea of doing so is to make the code more composable, where you can pass a function into other functions to perform and compose multiple operations together.

```
final List<Integer> numbers = Arrays.asList(1, 2, 3);
final List<Integer> numbersPlusOne =
    numbers.stream().map(number -> number + 1).
                collect(Collectors.toList());
```

Many languages, which are otherwise not supportive of Functional Programming, have functions as first-class citizens, including JavaScript and Java 8. In the example above, the function is a lambda, where it only exists within the context of its call site. Languages that support functions as first class citizens allow you to define the function outside of this context as a *function value*, and then use it where you see fit. In Python, we could do this like so:

```
>>> def addOne(x):
...     return x + 1
```

```
...
>>> myFunction = addOne
>>> myFunction(3)
4
```

## 3.3 Responsiveness to Users

Beyond functional programming, we also need to utilize tools that give us responsiveness to build Reactive applications. This is not Responsive Web Design[34], as it is known in the user experience world, such as writing a front end to be "responsive" to resize and move around seamlessly for the user. Instead, it is about being able to quickly respond to user requests in the face of failure that can occur anywhere inside or outside of an application. The axiom we use to define the tradeoffs we make when factoring the performance of a Reactive application is that you can choose any two of these following three characteristics, which will be defined further along in this section:

Footnote 34 http://en.wikipedia.org/wiki/Responsive_web_design

- High Throughput
- Low Latency, but also smooth and not jittery
- Small Footprint

### 3.3.1 The Cost of Abstractions

One of the hallmarks of Functional Programming is the ability to create abstractions – constructs of code that allow developers to provide similar functionality across many different concrete implementations. Abstraction aids developers in limiting the amount of duplicate code, as well as a higher level of reasoning about the code that they write. Abstractions can also make application programming interfaces (APIs), the way that external users leverage our functionality, simpler for them to use. Furthermore, abstractions can allow programmers to ensure a higher level of correctness in their code.

Abstractions can come at a cost, though. By providing a mechanism to reason about how to solve problems at a higher level, they can also reduce the throughput and increase the latency of an application. The higher the level of abstraction, it is plausible that the code is further removed from the "metal," or the way it will actually be executed by the computer. Even for higher-level languages that are run natively by an operating system and computer, such as C, there are already several layers of abstraction in place:

- The operating system which abstracts over the hardware
- Macro instructions that are emitted by the compiler are abstractions over the micro instructions to be executed by a core at runtime

If you run on a virtualized platform (such as the JVM for Java, V8 for JavaScript,

Rubinius or YARV for Ruby, Beam for Erlang, etc) in a virtual environment such as the cloud, the levels of abstraction increase even further:

- The instructions emitted by the compiler are abstractions to be converted to macro instructions for the specific platform on which they are executed
- The virtual machine of the cloud platform is an abstraction over the physical resources of the specific platform on which they are hosted

The more levels of abstraction that exist between a programmer and the physical hardware that will execute their platform, the greater the cost there is in translation. As developers add abstractions to their applications, they must be cognizant of the cost of those abstractions and the impact they may have on the program's ability to run be responsive at the time the program is executed.

### 3.3.2 Throughput

Throughput is a measurement of the maximum number of requests a service or application can handle within a specific time period, such as one second. For example, a web server may be able to handle as much as 100,000 requests in a second, assuming it isn't doing anything with the requests beyond returning an HTTP 200 (OK)[35] message. This is, in essence, a "no-op" or null operation that does no work of consequence other than measure the throughput of the web server itself. However, once you begin to put actual request handling work behind the server's interfaces, you will likely see a significant decrease in the throughput as the server has to do more work to calculate a value or update a data store of some kind.

Footnote 35 http://www.w3.org/Protocols/HTTP/HTRESP.html

Many companies shy away from publishing numbers about their ability to handle a specific number of requests per second, because the community at large will frequently ridicule the numbers and claim outlandish throughput of their own. When Twitter engineers publicly stated that they handle an average of over 5,000 tweets per second, many derided the number, claiming that this was evidence that Twitter does not have significant scale. Some posted numbers of how they had handled hundreds of millions of events and/or messages per second in their own benchmarks.

However, this does not take into consideration the work that Twitter must do or the guarantees they must make once a Tweet is received, including persistence, acknowledgement to clients, separating direct messages from public tweets, updating timelines, pushing notifications to followers, etc. Some of this is likely done asynchronously, and some is likely not. Furthermore, when Twitter's maximum measured load (as of the writing of this book) occurred during the debut of a Japanese

television show in 2013, they reached 143,000 messages per second without failure. This is the sign of an application that is able to withstand traffic that can arrive in bursts without having the notorious Twitter "Fail Whale" show up due to overload.

While contention for shared resources is the greatest concern for developers trying to make an application fast, another major factor is the amount of acknowledgement that must take place between various components and subsystems in order to guarantee receipt of messages and events. The more that an application must acknowledge receipt and ensure that such messages and events have been propagated successfully and in the right order, the more cost you will pay in your ability to handle events. Think deeply about the guarantees your application absolutely must make versus the guarantees that are merely nice to have.

Improvements in throughput frequently involve making the data that needs to be utilized at the time of execution more local. Local can have multiple meanings, however – data can be more local in space, as in data that will be used sequentially should be located closer together physically in caches so that it can be pre-fetched or retrieved quickly for usage one after the other, or it can be local in time, where data that is cached after use will be used again shortly thereafter. Organizing cached data, or data that is replicated somewhere nearby to make it easily accessible for repeat usage, depends on one of these two factors. When building an application, developers should consider the cost of caching based on locality in time or space in order to derive the most value from doing so.

### 3.3.3 Latency

Latency is the maximum time it takes from the time a request is received until it has been handled and the response sent. As stated in the previous chapter, it makes no sense to measure the "average" latency of a service in handling requests; instead we need to determine the distribution of latencies and look at percentiles of it, preferably for several different load scenarios. For example, a developer could measure latency at a throughput of 5,000 events per second at the 90th percentile, 95th percentile, 99th percentile and 99.99th percentile, and the same at 10,000 events per second, as well as 20,000 events per second. This is done to get an accurate picture of how increasing throughput impacts the latency of an application at varying levels of probability.

Latency can come from several different places, particularly in a distributed application, including:

- Network latency due to heavy traffic
- Data storage and persistence infrastructure can be overwhelmed and become unresponsive
- Cache coherency at the core, socket, main memory and external cache levels
- Equipment failure can occur

More and more applications are being built leveraging micro services, the architectural concept of breaking your application into multiple small services responsible for only a single task or resource, as opposed to monolithic services that do many different things. This way, such services can be utilized by multiple other services without duplicating logic and providing a very high level of separation of concerns[36]. Latency can have a nasty domino effect in such an environment. If a data store leveraged by multiple micro services begins to experience high load and becomes slower, the impact on the services calling them is immediate. However, services that call those services are now also affected, all the way upstream to the service attempting to respond to a client request. Applications that are built in this architectural fashion must consider the implication of latency downstream and how they will respond to clients.

Footnote 36 http://en.wikipedia.org/wiki/Separation_of_concerns

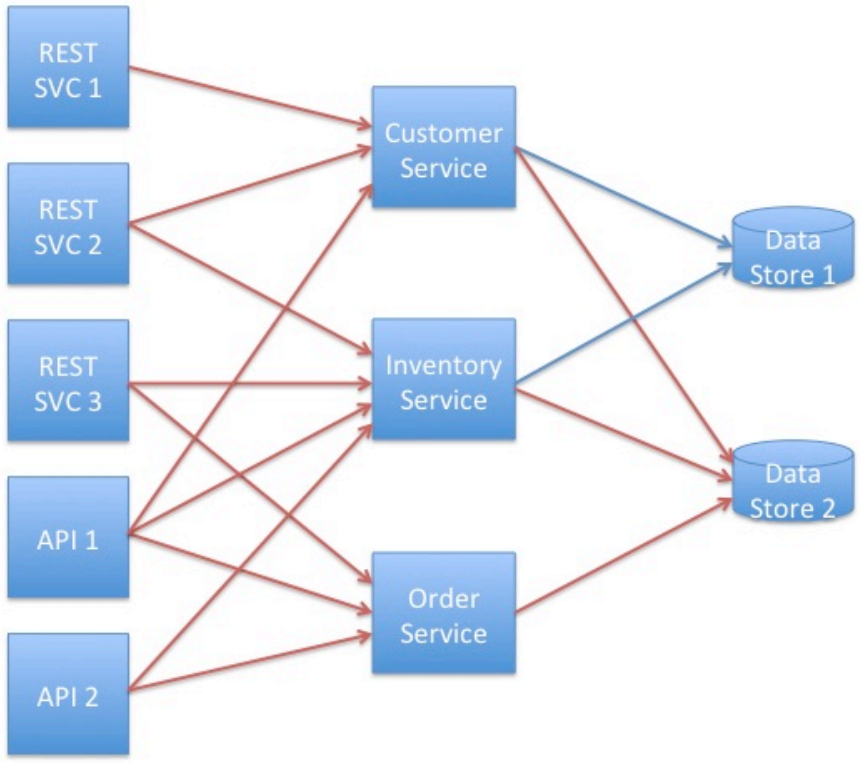


**Figure 3.2 Illustration of the effect of latency in a system composed of multiple services. If interaction with Data Storage 2 results in higher latency due to overload, all three services are affected. But the REST layers and API clients to those services are affected at an increasingly higher level.**

### 3.3.4 Footprint

The concept of footprint relates to the amount of resources used to be Reactive to users. This must be considered across several different kinds of resources, including:

- Deployed application artifact, and how small or large it is. The larger it is, the more time it can take to load the application at startup.
- Number of cores required, and how many your application must leverage to meet its performance goals.
- Memory, the amount of memory your application must use at runtime. The more you use, the more you can have issues with retrieval for usage at a core of execution, allocation costs, and if the application is deployed on a managed runtime such as the Java Virtual Machine, there could be increased costs with respect to garbage collection and compaction.
- Disk space, the amount of storage required
- Number of machines, as was shown in the example above when deploying a non-scaling application.
- Number of data centers, as an application may have to scale to serve clients around the globe while maintaining resilience.
- Amount of energy, which is more of a hidden cost in that it may not obvious at the time of design. As your footprint grows, the amount of energy and the monetary and environmental cost associated with it also goes up.

### 3.3.5 Prioritizing the Performance Characteristics

When we make architecture choices in building a Reactive application, we are in essence prioritizing two of the three of these as more important than one other. Note that this is not a law or theorem, but more of a guiding principle that is likely to be true. To get a very fast application with smooth, low latency, we typically have to give our application more resources (footprint). An example of this is the Disruptor Framework[37], which is a high performance messaging library created in Java. To get the tremendous throughput and smooth latency that they do, they have to pre-allocate all of the memory they intend to use for the internal ring buffer, and then reuse it to avoid intermediate allocations at runtime that could lead to stop-the-world garbage collection and compaction pauses within the Java Virtual Machine process in which it runs. The Disruptor also derives throughput by pinning to a specific core of execution on a box, to avoid the cost of context switches[38] between scheduled threads being swapped in and out on that core, which is another impact of footprint in that there is now one less core of execution on that box that is no longer available to any other threads to use.

Footnote 37 http://lmax-exchange.github.io/disruptor/

Footnote 38 http://en.wikipedia.org/wiki/Context_switch

The Storm Framework[39] was created by Nathan Marz and release in 2011 to much

acclaim, providing an out of the box capability for distributed handling of streaming data. The framework was created at Marz' startup, BackType, which was purchased by Twitter and became the basis for real-time analytics applications at the company. However, the implementation was not particularly fast, as it was built using Clojure and pure functional programming constructs. When Marz released version 0.7 of the framework in 2012, he used the Disruptor framework to increase throughput of the framework by as much as three times, at the cost of footprint. This matters to those who choose to deploy the framework as well, particularly in the Cloud, where one core on the VM must be used just for the Disruptor to maintain its speed. Given that the number of cores available to an application in a virtual environment is not an absolute value. As Doug Lea[40], the author of many concurrency libraries on the JVM such as the ForkJoinPool and CompletableFuture, has said, "Ask your hypervisor how much memory you have, or how many cores you have. It'll lie. Every time. It's designed to lie. You paid them money so it would lie."[41] Developers have to take these variables into account when considering footprint in a cloud deployment.

Footnote 39 https://github.com/nathanmarz/storm

Footnote 40 http://en.wikipedia.org/wiki/Doug_Lea

Footnote 41 http://chariotsolutions.com/screencast/phillyete-screencast-7-doug-lea-engineering-concurrent-library-components/

Other platforms have constraints that limit their ability to make these tradeoffs. For example, a mobile application typically cannot give up footprint in order to increase throughput and make their latency smoother, because the amount of resources on the mobile platform is more limited than a server. Imagine the cost of pinning a core on a mobile phone, both in terms of reduced resource availability for other applications and the phone operating system. There is also less memory resources, as well as constraints about power usage as you don't want to drain the battery on the device so quickly that no one would want to use your application. Instead, mobile applications typically attempt to minimize footprint while increasing throughput at the expense of latency, as users are more willing to accept latency on a mobile platform. Anyone who has used an application from a phone has experienced slowness due to network issues or packet loss.

### NON-FUNCTIONAL REQUIREMENTS

Making decisions about the tradeoffs between throughput, latency and footprint requires that the designers of the system to be built first think through what non-functional requirements (NFRs) they consider to have the highest priority. No application should be built without decisions being made about these requirements, as they allow the developer to frame the problem that must be solved. Beyond throughput, latency and footprint, these non-functional requirements may include:

- Uptime and availability of the service
- Disaster recovery in the face of unexpected failure
- Compliance with various standards and organizational protocols
- Security
- Configuration
- Resilience

### ASYNCHRONOUS VERSUS SYNCHRONOUS

Responsiveness has another dimension that must be considered. While Reactive applications are built using asynchronous messages and by being event-driven, there are times when responsiveness is dependent on performing a task synchronously. Asynchronous execution does not necessarily guarantee speed. There may be tasks that can best be performed via synchronous interactions due to the effect of Amdahl's Law[42] (as discussed in Chapter 1), where your ability to parallelize is limited by the cost of any actions in the operation that must be sequential.

Footnote 42 http://en.wikipedia.org/wiki/Amdahl's_law

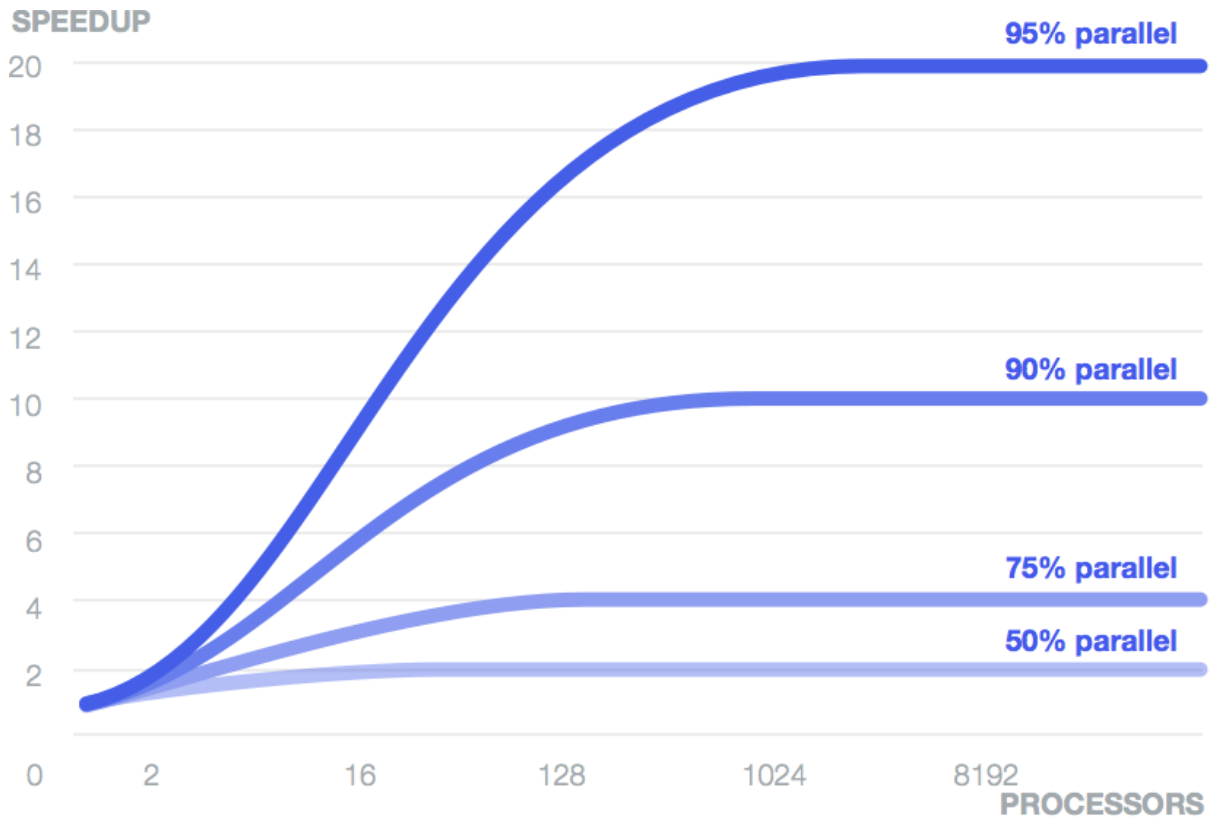


**Figure 3.3 Amdahl's Law**

As an example of the cost of Amdahl's Law, consider parallel collections. The concept is powerful, that merely by the act of defining a collection as parallel, or converting a non-parallel collection to one that is, the work to be applied to the elements

in such a collection can be spread across all of the cores on a machine in which the operation is being performed. This sounds like an easy way to leverage parallelism, and it is, but it is not a panacea. A task that will be applied to a parallel collection must be commutative (where order does not matter) and associative (where grouping does not matter). The tasks can then be recursively broken down into sets of data that will be performed by each core on the computer. However, the cost of joining the data at the end of the operation, which is sequential in nature, may be more expensive than had the entire operation been performed on a single core sequentially in the first place. Developers must benchmark such tasks to see if a performance gain is realized and not accept that parallelism is the key to performance as dogma.

Consider a list of integers in a collection, representing values from 1 to 1 million. To add one to each integer, a function that takes each value x and adds one to it yielding a new value of x + 1 can be applied commutatively and associatively – the order in which the function is performed to each element (commutativity) has no bearing on any other in the collection, and the grouping of values is irrelevant as well (associativity), and therefore is a good candidate for being parallelized. The implementation of the parallel collection can recursively break down the integers in the collection to unrelated groups where the application of the function can be done in any order. But the cost of reassembling the list into a single list result value, as well as the additional allocation of the new elements in the list, may be greater than had the work been done synchronously.

Also, consider the axiom of throughput, latency and footprint in this case. To utilize a parallel collection, we are increasing our footprint in space (allocations) and number of cores. It is more difficult to say we are increasing our throughput or lowering our latency and footprint without benchmarking the implementations relative to a sequential, single-threaded implementation. A developer may actually make the algorithm worse across all three dimensions by performing an action without considering the costs associated with it. In this case, well-written micro benchmarks for the platform are essential for making design decisions that can aid in your responsiveness. While there are some assumptions we can make about performance in our designs, it is frequently best to measure various approaches first and make decisions based upon the results.

## *3.4 Implementations That Support Reactive*

Now that the basic concepts of how to consider Reactive tooling has been reviewed, it is time to look at the various constructs and tools that exist in languages today support these ideals. Many languages have innovated in the area of dividing work and making it asynchronous. It is time to investigate how that is currently done by existing tools.

For each of the implementations below, a discussion of how well they meet the tenets of the Reactive Manifesto is provided. Note that while all of them are asynchronous and non-blocking implementations themselves, it is up to the developer to ensure that the

work they do remains asynchronous and non-blocking as well to remain Reactive.

### 3.4.1 Green Threads

In some languages, the capability to utilize multiple threads does not exist as a core construct. In these cases, it is still possible to leverage asynchronous behavior via green threads[43], which is the capability to hand off a thread of execution between multiple branches of logic to be shared, as the thread scheduling is managed in user space and therefore does not interact with the platform's operating system.

Footnote 43 http://en.wikipedia.org/wiki/Green_threads

Green threads provide the capability to be very efficient in the usage of threads in an application, but are restricted to a single physical machine. It is impossible, without the aid of delimited, portable continuations[44], to share the processing of a thread across multiple physical machines. These continuations allow you to mark points in logic where the execution stack can be wrapped up and exported either locally or to another machine. Such continuations can then be treated as functions, which is a powerful idea, and was even implemented in a Scheme library called Termite[45]. But green threads and continuations do not provide for resilience as there is currently no way to supervise their execution, and are therefore lacking with respect to fault tolerance.

Footnote 44 http://en.wikipedia.org/wiki/Delimited_continuation

Footnote 45 https://code.google.com/p/termite/

As stated in the previous chapter, the paper A Note on Distributed Computing by Waldo et al[46] notes that it is not a good idea to try to make logic that executes in a distributed context appear to be local, and this postulate can be applied to this scenario as well. If the local is considered to be local to the thread, and distributed/remote is considered to be on another thread and thus asynchronous, a developer would not want asynchronous operations appear to be synchronous because hiding the semantic difference obscures the impact of the way each are executed.

Footnote 46 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

#### REACTIVE CLASSIFICATION OF GREEN THREADS

Green threads are asynchronous and non-blocking, but do not support message passing. They do not scale up to use multiple cores on a machine by themselves, though if the runtime supports it, it is possible to have more than one in a process or multiple processes can be run. They do not scale outward across nodes. They also do not provide any mechanisms for fault tolerance, and it is up to developers that use them to write their own constructs to handle failure that may occur.

### 3.4.2 Event Loops

When a language or platform does not support multiple threads in a process,, it is still possible to leverage asynchronous behavior via the green threads mentioned above that share an event loop. This loop provides the mechanism for sharing this single execution thread among several logical threads at the same time. The idea is that while only a single thread can execute at a given moment, the application should not block on any operation and should instead yield the thread until such time as the external work it needs, such as calling a data store, is completed. At that point in time, a callback can be performed representing the pre-defined behavior to be applied. This is very powerful – Node.js[47] allows a single-threaded language such as JavaScript to perform considerably more work because it doesn't have to wait for every operation to complete before handling other work.

Footnote 47 http://nodejs.org

While Event Loops allow languages with no ability to be multithreaded to have asynchronous capabilities, the implementation of the event loop handlers is most typically done via callbacks. That would be okay if only a single callback could be referenced at a time, but as an application's functionality grows, this is typically not the case. While tools such as Node.js have tremendous popularity in a large segment of the programming population, the terms "callback hell" and "pyramid of doom" have been coined to represent the interwoven spaghetti code that often results from its usage. Furthermore, event loops based on a single-threaded process are only viable for uses that are I/O bound, or when the use case is specific to handling input and output. Trying to use an event loop for CPU bound operations will remove the advantage of the this particular approach.

Here is a simple example of a Node.js application. Note that running this server and using Google's Chrome browser to send requests to the 127.0.0.1:8888 address may result in a doubling of the counter value on each request. Chrome has a known issue with sending an additional request for a *favicon* with every request.[48]

Footnote 48 Chrome issue report: https://code.google.com/p/chromium/issues/detail?id=39402

```
var http = require('http');
var counter = 0;

// Set up the server, with a callback function applied
// to respond to each request with a message in the body
http.createServer(function (req, res) {
  counter += 1;
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Response: ' + counter + ' via callback\n');
}).listen(8888, '127.0.0.1');
```

```
// Let the user know the server is
// running and on which URL/Port
console.log('Server up on 127.0.0.1:8888, send requests!');
```

#### REACTIVE CLASSIFICATION OF EVENT LOOPS

This depends on the implementation. When deployed via Node.js in JavaScript, event loops are similar to green threads, in that they are asynchronous and non-blocking, but do not support message passing. They do not scale up to use multiple cores on a machine by themselves, though if the runtime supports it, it is possible to have more than one in a process or multiple processes can be run. They do not scale outward across nodes. They also do not provide any mechanisms for fault tolerance, and it is up to developers that use them to write their own constructs to handle failure that may occur.

However, there are alternative implementations such as Vert.x[49], which runs on the Java Virtual Machine and has a very similar feel to Node.js, but supports multiple languages. Vert.x is a compelling solution because it provides a distributed approach to the event loop model, using a Distributed Event Bus to push messages between nodes. In a JVM deployment, it does not need to leverage green threads as it can use a pool of threads for multiple purposes. In this respect, Vert.x is asynchronous and non-blocking, and does support message passing. It also scales up to use multiple cores, as well as scaling out to leverage multiple nodes. Vert.x does not have a supervision strategy for fault tolerance, but is an excellent alternative for an event loop solution, particularly because it supports JavaScript as Node.js does.

Footnote 49 http://vertx.io/

### *3.4.3 Communicating Sequential Processes*

Based on a paper written by Tony Hoare and others in the late 1970s, Communicating Sequential Processes (CSP) are the idea that multiple processes, or threads inside of a single process, can communicate via message passing. Such message passing enabled asynchronous capabilities, as a developer can define work in each process or thread concurrently and then pass messages between them to share information.

What makes CSP unique is that the two processes or threads do not have to know anything about one another, which means they are nicely decoupled from a sender and receiver standpoint, but still coupled with respect to the value that is actually being passed. There also are no queues between the sender and receiver of the message—the sender and receiver must reach a point where the sender is ready to send and the receiver is ready to receive for the message to be passed. This is fundamentally different from Actors, which will be discussed in a later section. It also means that the two processes or threads may limited in how distributed they can be, depending on how CSP is

implemented. For example, CSP on the JVM via Clojure's core.async library cannot be distributed across multiple JVM instances, even on the same machine. Neither can Go's channels, also known as *goroutines*.

One particularly intriguing aspect of CSP in the formal sense is that the logic of how the processes communicate is mathematical in nature, and therefore theoretically *provable* that a deadlock can or cannot occur inside of it via Process Analysis. Being able to statically verify this level of concurrent logic correctness is a very powerful idea. Note that neither Clojure's core.async nor Go's channels have this capability, but if it is practical to implement, it would be very useful.

Since no process or thread in a CSP-based application has to know about one another, there is a form of location transparency—the developer does not have to know anything about the other process or thread with which it will be communicating. However, due to the limitations of not being able to write such channels between runtimes, true location transparency cannot currently exist in the most popular implementations of CSP to date. There is also a question of fault tolerance, as failure between two processes or threads cannot be managed easily. Instead, the logic in both processes or threads must have the ability to manage any failure that could occur in communicating with the other side. Another potential downside is that the any non-trivial implementation of CSP could be difficult to reason about, as every process/thread can potentially interact with every other process/thread at each step.

Here is a simple example of two communicating processes in Go. Interestingly, a Go function can create a Channel, put values onto it and consume them as well, practically stashing values off to the side for use later on. In this example, there is a function that will produce messages and a second function that will consume them from the Channel.

```
package main

// Import required language libraries
import (
    "fmt"
    "time"
)

// Define the driver for the application, creating the
// number of iterations and the channel, and then calling
// the asynchronous functions that will produce and consume
// the values on that channel.  Sleep so that the program
// does not exit immediately.
func main() {
    iterations := 10
    myChannel := make(chan int)

    go producer(myChannel, iterations)
    go consumer(myChannel, iterations)
```

```
    time.Sleep(500 * time.Millisecond)
}

// Define producer behavior, iterating however many times
// and putting the count of the iteration into the channel
func producer(myChannel chan int, iterations int) {
    for i := 1; i <= iterations; i++ {
        fmt.Println("Sending: ", i)
        myChannel <- i
    }
}

// Define consumer behavior, iterating the same number of
// times and getting the value off of the channel
func consumer(myChannel chan int, iterations int) {
    for i := 1; i <= iterations; i++ {
        recVal := <-myChannel
        fmt.Println("Received: ", recVal)
    }
}
```

#### REACTIVE CLASSIFICATION OF CSP

CSP is asynchronous and non-blocking and does support message passing. They do scale up to use multiple cores on a machine. They do not scale outward across nodes in any of the current implementations. They also do not provide any mechanisms for fault tolerance, and it is up to developers that use them to write their own constructs to handle failure that may occur.

### *3.4.4 Futures and Promises*

A Future is a read-only handle to a value or failure that will become available at some point in time. This means that a programmer can define an operation in code that will be executed by some thread at some point in time, and a value may or may not be returned from the evaluation of the code inside of it. There is no way for a developer to programmatically set a value inside of a Future aside from the expected return of a value at the end of it, and therefore they can be treated as immutable, whereas a Promise is a value that the developer can attempt to set themselves, and if the value inside of the Promise has not already been set, it will successfully complete the Future inside of it. Note that these definitions for Future and Promise are those of the authors, and different languages and platforms may use different terminology to represent these concepts.

Futures are a very simple way to make code asynchronous, as by merely defining the code to be executed as a Future, it could be run by the current thread if it has finished its current tasks before the work can be scheduled elsewhere, or it will be run by another thread concurrently. Futures will return either the result of their successfully evaluation, or a representation of whatever error may have occurred during that evaluation.

An elegant example of the usage of Futures can be found in the retrieval of data from

multiple sources at the same time. Imagine a service that needs to return Customer information from a data store – these data are stored in a database somewhere, but they may also be cached in a store that is closer for performance reasons. To retrieve the data, a programmer should check the cache to see if the more local store has the data needed to avoid an expensive database lookup. But if there is a cache miss, where the data to be retrieved from the cache is not found, the programmer must now look in the database to find the data to be returned to the client.

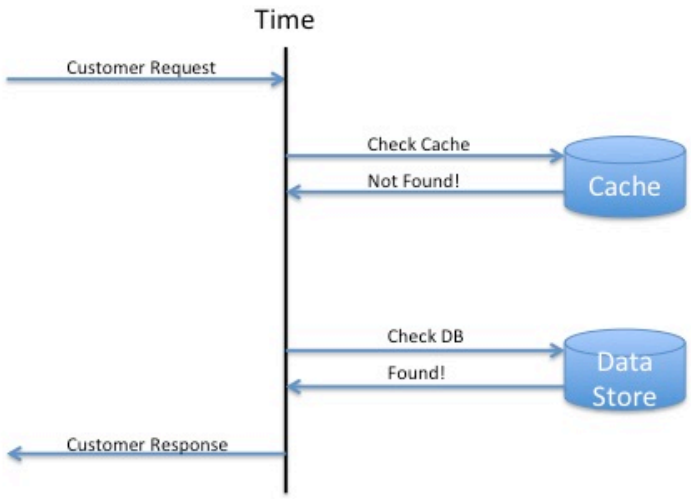


**Figure 3.4 Illustration of sequential lookup, where a cache is checked first, and if no data is found a call to retrieve the data from the database is made.**

In the sequential lookup case, when the cache miss occurs, the calling thread must then make an additional call to the database to retrieve the customer data. When that returns, the response is sent back, but at the cost of two lookups that took place one after the other.

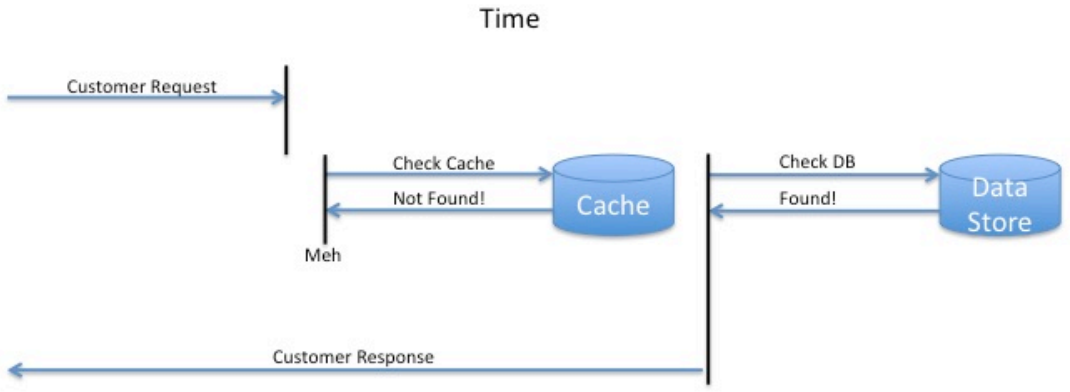


**Figure 3.5 Illustration of a parallelized lookup, where calls are made to the cache and database immediately, and whichever returns a value first results in the response to the client.**

In this parallel lookup case, both requests to the cache and the database are sent simultaneously. If the cache responds with a found customer record first, the response is sent back to the client immediately. When the database responds with the client information, it is ignored. However, if the cache miss occurs again, the calling thread doesn't have to make a subsequent database lookup call, as that has already been done. When the database responds, theoretically quicker than if the client had made sequential calls, the response is sent to the client right away.

The code for this task would look similar to this, written in Java 8 to leverage the new non-blocking CompletableFuture functionality it provides:

```
public class ParallelRetrievalExample {
    final CacheRetriever cacheRetriever;
    final DBRetriever dbRetriever;

    ParallelRetrievalExample(CacheRetriever cacheRetriever,
                             DBRetriever dbRetriever) {
        this.cacheRetriever = cacheRetriever;
        this.dbRetriever = dbRetriever;
    }
```

```
    public Object retrieveCustomer(final long id) {
        final CompletableFuture<Object> cacheFuture =
           CompletableFuture.supplyAsync(() -> {
                    return cacheRetriever.getCustomer(id);
                });
        final CompletableFuture<Object> dbFuture =
           CompletableFuture.supplyAsync(() -> {
                    return dbRetriever.getCustomer(id);
                });

        return CompletableFuture.anyOf(
            cacheFuture, dbFuture);
    }
}
```

Performing these two operations sequentially would be expensive, and there is very rarely an opportunity to cache data beforehand without knowing what a client will ask for next. As a result, using Futures to perform these two tasks is a very handy tool to perform them in parallel. Using Futures, a programmer can create two tasks to look for the data in the cache as well as the database virtually simultaneously, and whichever of the two tasks completes first is included in the response to the client who requested the data.

The act of performing such a concurrent lookup does help a programmer leverage more resources (footprint) in order to reduce the time to complete a response when a request is received (latency), but there is a plausible scenario when neither the cache lookup nor the database retrieval will happen within the non-functional requirements of the service for which they are performed. In this respect, any Future implementation should also have a timeout mechanism allowing the service to communicate to a client that the operation is taking too long, and that they will either need to make another attempt to request the data or communicate downstream that there is a failure taking place within the system. Without timeouts, the application cannot be responsive to a user about what is happening and allow them to decide what they should do about it.

Futures are not non-blocking by definition, and can vary by implementation. For example, Java prior to Java 8 had a Future implementation, but there was no way to get the value out of the Future without blocking in some fashion. A developer could write a loop that calls the isDone() method on one or more Future instances to see if they were completed, or they could call the get() method which would block until the Future failed or completed successfully. Make certain that the Future implementation in a version of a language is non-blocking before considering whether to use it, and if not, consider alternatives that do not pay this penalty.

Similarly to the Event Loops described above, Futures can be handled with callbacks, allowing a developer to pre-define logic to be applied upon the completion of the Future.

But, like Node.js, the usage of callbacks can quickly turn ugly when more than one is applied at a time. Some languages that support Functional Programming allow a user to map over a Future, which also pre-defines behavior that is only applied when the Future successfully completes, not if it fails. By using a higher-order function[50] such as map, these languages frequently have the ability to compose the behavior upon completion of many Futures into simple, elegant logic using syntactic sugar called a for or list comprehension[51], which is particularly useful when staging results from multiple Futures into a single result.

Footnote 50 http://en.wikipedia.org/wiki/Higher-order_function

Footnote 51 http://en.wikipedia.org/wiki/List_comprehension

```
/** Returns a Future of a tuple of the local inventory
 * and overall inventory for a specific product.  Both
 * requests are sent individually, but the Future to get
 * the tuple of the values cannot complete until both
 * values are returned.
 *
 * The local inventory will just be a count, but the overall
 * inventory is a Map of Warehouse ID to count.
 */
def getProductInventoryByPostalCode(
      productSku: Long,
      postalCode: String):
        Future[(Long, Map[Long, Long])] = {

  // Provide the thread pool and Future timeout value to
  // be applied. Because these are implicit variables, they
  // are automatically applied to each Future usage below
  implicit val ec = ExecutionContext.
                      fromExecutor(new ForkJoinPool())
  implicit val timeout = 250 milliseconds

  // Define the futures so they can start doing their work
  val localInventoryFuture = Future {
    inventoryService.currentInventoryInWarehouse(
      productSku, postalCode)
  }
  val overallInventoryFutureByWarehouse = Future {
    inventoryService.currentInventoryOverallByWarehouse(
      productSku)
  }

  // Retrieve the values and return
  // a future of the combined result
  for {
    local <- localInventoryFuture
    overall <- overallInventoryFutureByWarehouse
  } yield (local, overall)
}
```

As mentioned earlier, a Promise is similar to a Future in that they both define work that will be done asynchronously, but they are different in that a Promise represents a write-once value that may be completed by any one of several operations. In the CompletableFuture cache and database loading example, the capability to determine whatever Future finished first was utilized to return a value as the response to the user without capturing the value for local use. A developer can use a Promise create a Future to be returned that is not bound to some locally defined block of code. A Promise can also be used by a developer to create a "race" between multiple Futures to "complete" the promise, and because it is a write-once value, they can be sure that the first asynchronous task to place a value into the Promise is the result they will receive from the Future inside of the Promise.

There are alternative formulations for Futures and Promises that exist in some languages, such as the concept of Continuation Passing Style (CPS) and Dataflow that is based up on it. The beauty of this particular kind of construct is that it provides developers with a way to write code that appears to be synchronous, but is actually compiled into Futures. This is done because order can be maintained – a block of code can be written to occur before another block of code will be performed. Despite the asynchronous nature of the code resulting from Dataflow, the logic is still deterministic (as long as it is free of side effects, as described earlier in this chapter), meaning that each time it is executed it will behave in the same way. If the application were to enter a deadlock state in the Dataflow code one time, it will do so every time because the evaluation will remain ordered the same way.

There is an emerging concept of a "safe" Future, where the methods that can be executed concurrently are annotated or marked some such fashion, merely providing the runtime with the option of doing so if it sees an opportunity for optimization by parallelizing parts of the code where no data is shared. This is a compelling idea, but still subject to potential errors when someone accidentally exposes data to be shared in a method marked as safe. Furthermore, it provides no oversight for failure. Futures in general are a very narrow abstraction, in that they allow you to define a single operation that will take place off thread one time and need to be treated as a single unit of work. They also do not handle resilience particularly well, where you have to use Future implementations that communicate what went wrong when failure occurs on the thread of execution.

#### REACTIVE CLASSIFICATION OF FUTURES AND PROMISES

Futures and Promises are asynchronous and non-blocking, though they do not support message passing. They do scale up to use multiple cores on a machine. They do not scale outward across nodes in any of the current implementations. They do provide mechanisms for fault tolerance at the individual level, and some languages may aggregate failure across multiple Futures such that if one of them fails, all of them do.

### 3.4.5 Reactive Extensions

Reactive Extensions[52] originated in the .NET world, devised and built by Erik Meijer and his team at Microsoft. The idea was to combine the functionality of two control flow patterns—an iterable and the Observer pattern[53]. In the two cases, both involve handling a potentially unknown number of items or events. In the case of a synchronously executed iterable, the developer can write a loop construct to get each item individually and perform work on it, always in control of when the work is to occur. In the asynchronous Observer pattern, a developer registers a callback to be used when an event occurs.

Footnote 52 https://rx.codeplex.com/

Footnote 53 http://en.wikipedia.org/wiki/Observer_pattern

Reactive Extensions take the two ideas and combine them into an Observable, where the developer writes a looping construct that reacts to events that occur elsewhere. This is similar to *streaming* semantics, where data endlessly iterated over as it arrives for processing. The library has extensions built in that allow for composing the application of functions to the data using standard operators such as filter, accumulate or even perform time-sensitive functions on several such events. Where Futures asynchronously return a single value to be handled, Observables are abstractions over streams of data that can be handled in groups. Observables also have the capability to tell a consumer when they are finished, much as an iterator could.

However, there are a few drawbacks to the base implementation. Similar to Futures, failure handling can be problematic, in that there is no overall supervision that oversees what happens to events once they are processed—there is only a handler method you can implement. There is also concern about *backpressure*, or the ability to communicate backwards through the system that the application is saturated in load. If you need the ability to implement backpressure, you must do so yourself. And the same composability issues that existed with Iteratees are in play with Rx—if the application has a partial state that was gleaned from one event and then must be completed by another, it can be difficult to build the data if the events aren't ordered or failure occurs.

Observables are created from a source of some kind, which can be literals, collections or even a socket. A subscriber provides the handler functions for what to do when a

chunk of data is to be processed or when an error occurs. An example of an RxJava Observable to handle streams could look like this:

```
package org.reactivedesignpatterns.chapter2.rxjava;

import rx.Observable;
import rx.functions.Action1;

// Create the class with the Observable, reacting to the
// events that it will consume
public class RxJavaExample {
    RxJavaExample() {
    }

    public void observe(String[] strings) {
        Observable.from(strings).subscribe((s) -> {
                System.out.println("Received " + s);
        });
    }
}
```

And an example of a driver that would produce the events consumed by that Observable might look like this:

```
package org.reactivedesignpatterns.chapter2.rxjava;

public class RxJavaExampleDriver {
    final RxJavaExample rxJavaExample = new RxJavaExample();

    public static void main(String[] args) {
        String[] strings = { "a", "b", "c" };
        rxJavaExample.observe(strings);
    }
}
```

## REACTIVE CLASSIFICATION OF REACTIVE EXTENSIONS

Reactive Extensions are asynchronous and non-blocking, and while the call into logic you create, they do not directly support message passing. They do scale up to use multiple cores on a machine. None of the current implementations scale outward across nodes, though the Akka ReactiveStreams implementation will include this functionality. They also do not provide any mechanisms for fault tolerance, and it is up to developers that use them to write their own constructs to handle failure that may occur.

### 3.4.6 The Actor Model

Early on in this chapter, the Erlang programming language was mentioned as one of the earliest implementations that supported Reactive application development. Erlang uses Actors as it's primary architectural construct, where communication between entities can only happen via message passing on the sending side and mailbox queues on the receiving side. Carl Hewitt introduced actors in a white paper in 1973, and therefore the concept has been around quite a long time. With the success of the Akka toolkit on the JVM, Actors have had a surge in popularity of late.

#### ASYNCHRONOUS

The definition of Reactive states that interactions should be event-driven, asynchronous and non-blocking, and actors meet all three of these criteria by definition. This means that the programmer does not have to do anything extra to make their logic asynchronous besides creating multiple actors and passing messages between them. The only thing the programmer must do is to avoid the use of blocking primitives for synchronization or communication within their actors, as these would negate the benefits of the Actor Model.

#### FAULT TOLERANCE VIA SUPERVISION

As mentioned in the earlier section on Supervision, Actors usually support the idea of organization into supervisor hierarchies to manage failure at varying levels of importance. When an exception occurs inside of an Actor, that Actor instance may be resumed, restarted or stopped, even though the failure occurred on a thread executing asynchronously. Erlang's Open Telecom Platform (OTP)[54] defined a pattern of building hierarchies of supervision for Actors, allowing a parent Actor to manage failure for all children below it, or provide the possibility for elevation of failure to an appropriate parent Actor where it could be managed for a whole branch.

Footnote 54 https://github.com/erlang/otp

What is particularly interesting about this approach to resilience is that now failure can become a part of an application's domain, just like classes that represent data to be persisted. When designing an Actor application, a developer should take the time to think of all of the ways the application can fail at all levels of the actor supervisor hierarchy, and what they should do about each of those kinds of failure. They can also consider how to handle failure that they cannot foresee, and allow the application to respond to that as well: even though the precise cause of the failure might not have been anticipated, it is always safe to assume that the failed component has reached an invalid state and needs to be discarded. This principle is called "let it crash" and allows the

formulation of responses to failure scenarios that were not explicitly planned for. Without supervision, this kind of resilience is not possible, and even if it were, would result in failure handling strewn about the logic of the application.

### LOCATION TRANSPARENCY

Because Actors in Erlang and Akka provide a proxy through which all Actor interactions must take place (a PID in Erlang, and an ActorRef in Akka), individual Actors do not need to know the physical location of another Actor to whom they want to send a message. This has one benefit of making message sending code more declarative, because it does not involve any work beyond merely sending the message. All of the physical "how-to" details of how the message is actually sent is dealt with behind the scenes. This location transparency also means that if an Actor on one node does not exist any more, it is plausible that, if the cluster of nodes is smart enough, another Actor will be created on another node and messages are automatically routed to that one.

One of the larger concerns about Actors is that producers and consumers of messages are coupled to one another – the sender of the message must have a reference to the Actor instance to whom they want to send the message. This reference is equally necessary as a recipient address on a letter, without it the mail service would not know whither to deliver it. An Actor reference shares much of the characteristics of a postal address in that it merely describes how to transport messages, but not how the person looks like or what state it is in. Since actors only deal in such references, they also are less coupled when failure occurs. On the other hand, location transparency also means that the actor which is responsible for handling failure—its supervisor—is also protected by this isolating layer of message passing.

### BEHAVIOR CHANGES

Actor implementations have a very unique capability in that they can redefine their behavior in response to a message, which enables them to change the set of messages they listen to on the fly. Imagine an Actor that does not handle any messages except *Start*, and then when it receives a *Start* message, it switches behavior to handle those messages that allow it to do specific kinds of work and ignore any *Start* messages received after that point. It is unique, because the Actor is literally changing its API at runtime and we will discuss uses of this feature in chapter nine. This is a very powerful way to construct Finite State Machines[55] inside of Actors.

Footnote 55 http://en.wikipedia.org/wiki/Finite-state_machine

### NO CONCURRENCY

Because Actors only allow a single thread of execution within themselves, and no thread can call into them directly, there is no concurrency inside of the Actor instance themselves unless a programmer accidentally exposes something. This means that Actors can encapsulate mutable state and not worry about requiring locks to limit simultaneous access to variables.

This greatly simplifies the logic inside of Actors, but it does come at some cost. Actors can be implemented in two ways—as heavyweight, thread-based Actors which have a dedicated thread assigned to them, or as lightweight, event-driven Actors that share a thread pool and therefore consume less memory footprint. Regardless of the implementation, a concept of *fairness* has to be introduced, where the programmer defines how many messages an Actor will handle before giving up its ability to use its thread on a particular core before yielding to another Actor. This must be done to prevent *starvation*, as no one Actor should use a thread and/or core so long that other Actors cannot do their own work. Even if a thread-based Actor is not sharing its thread with other Actors, it most likely is sharing cores of execution at the hardware level.

### DIFFERENCES BETWEEN ERLANG AND AKKA

Given that there are these two prevalent Actor libraries in existence, how is a developer to choose which one is more appropriate for their application? This boils down to their application requirements and the platforms on which the two implementations are built. In the case of Erlang, the Beam VM allows each actor to be implemented as a distinct and isolated process, and this is a fundamental reason for the remarkable claims of fault tolerance in Erlang applications.

Erlang Actors use a pattern called *Selective Receive*, where an Actor receives a message and determines whether or not it is able to handle that message at a given time. If the message cannot be handled, it is put aside for the time being and the next message is handled. This continues until a message is received that can be handled by the current receive block of the Actor, at which time it processes that message and then attempts to retry all messages that were put aside. This is, in effect, a memory leak—if those messages are never handled, they continue to be set aside and replayed every time a message is successfully handled. Because the processes in the BeamVM are isolated, there is no theoretical impact if an Actor's process fails for exceeding maximum memory.

However, on the JVM, there is no such luxury. An exact port of Erlang and OTP on the JVM with Selective Receive would be a memory leak that would eventually, given enough time, take down the entire JVM with an OutOfMemoryError because the Actors

share the same heap. For this reason, Akka Actors have the ability to *stash* messages on demand, not automatically, and also provide the developer with the programmatic means to unstash and replay those messages at their leisure.

Here is an example of an Akka Actor application, with fault tolerance built in. In this case, there is a parent Actor that will supervise two children, each of whom will increment a value and assign it to a local counter. When the counter exceeds 1000, the actor that receives the value that is too large will throw a CounterTooLargeException, causing the supervising parent to restart the two actors, thus resetting their counters.

```
package org.reactivedesignpatterns.chapter2.actor

import akka.actor._
import akka.actor.SupervisorStrategy.Restart
import akka.event.LoggingReceive

case object Start
case class CounterMessage(counterValue: Int)
case class CounterTooLargeException(
  message: String) extends Exception(message)

class SupervisorActor extends Actor with ActorLogging {
  override val supervisorStrategy = OneForOneStrategy() {
    case _: CounterTooLargeException => Restart
  }

  val actor2 = context.actorOf(
                Props[SecondActor], "second-actor")
  val actor1 = context.actorOf(
                Props(new FirstActor(actor2)),
                "first-actor")

  def receive = {
    case Start => actor1 ! Start
  }
}

class AbstractCounterActor extends Actor with ActorLogging {
  var counterValue = 0

  def receive = {
    case _ =>
  }

  def counterReceive: Receive = LoggingReceive {
    case CounterMessage(i) if i < 1000 =>
      counterValue = i
      log.info(s"Counter value: $counterValue")
      sender ! CounterMessage(counterValue + 1)
    case CounterMessage(i) =>
      throw new CounterTooLargeException(
        "Exceeded max value of counter!")
  }

  override def postRestart(reason: Throwable) = {
```

```
    context.parent ! Start
  }
}

class FirstActor(secondActor: ActorRef) extends
  AbstractCounterActor {
  override def receive = LoggingReceive {
    case Start =>
      context.become(counterReceive)
      log.info("Starting counter passing.")
      secondActor ! CounterMessage(counterValue + 1)
  }
}

class SecondActor() extends AbstractCounterActor {
  override def receive = counterReceive
}

object Example extends App {
  val system = ActorSystem("counter-supervision-example")
  val supervisor = system.actorOf(Props[SupervisorActor])
  supervisor ! Start
}
```

## COMBINING ACTORS WITH OTHER CONSTRUCTS

If there is any one drawback to using Actors in Erlang and Akka, it is that they can be too heavyweight a construct to fit all problems. There are times when it is more appropriate to consider using a green thread solution, or CPS, or another construct, which will allow a single thread to perform multiple tasks. Actors handle a message on a thread, and while they can spawn Futures or other Actor instances to handle their work asynchronously, they cannot tell those other Actors and Futures to share a single thread and switch on blocking or on an event loop. This must be considered when a developer builds a Reactive application – use Actors for the architecture infrastructure to build a cluster and communicate between nodes and services, but there may be times to go with a lighter weight construct inside of those. And if you do, how is that thread inside of the Actor bound to the framework or platform? If you were to use an Actor's thread in Akka to try to do CSP, that would likely not work very well due to the *Inversion of Control*[56] nature of Akka's thread management. If a developer were to use another thread from a separate Dispatcher/Executor that wasn't linked to the runtime, the two paradigms could be combined effectively.

Footnote 56 http://en.wikipedia.org/wiki/Inversion_of_control

### REACTIVE CLASSIFICATION OF ACTORS

Actors are asynchronous and non-blocking and do support message passing. They do scale up to use multiple cores on a machine. They also scale outward across nodes in both the Erlang and Akka implementations. They provide supervision mechanisms as well in support of fault tolerance. They meet all of the requirements for building Reactive applications. This does not mean that Actors should be used for every purpose when building an application, but they can easily be used as a backbone, providing architectural support to services that leverage other Reactive technologies.

## *3.4.7 Summary*

We have now reviewed the essential concepts and constructs required for building Reactive applications and help us leverage the power of Fault Tolerance and Scalability to be Responsive to our users. In the next chapter, we will delve into the Reactive philosophy and discuss how these and other concepts relate to it. At this point, you should have a clear understanding about:

- The costs of building an application that is not Reactive
- What Functional Programming is and how it relates to Reactive applications
- The tradeoffs involved in choosing between high throughput, low/smooth latency and small footprint
- Pros and cons of all application toolkits that support the Reactive model
- How the Actor model simultaneously addresses both Fault Tolerance and Scalability

# II *The Philosophy in a Nutshell*

In the first part we have taken a comprehensive look at what it means for a system to be reactive, we have seen how the requirement to always react to user input entails resilience and scalability in order to retain responsiveness even during failures and varying load conditions. Throughout the exploration of these desirable properties we have encountered the need for the underlying implementation to be message-driven.

This part complements the description of the four reactive traits in that it presents a set of building blocks from which a reactive architecture can be constructed. Where part one described what we want to achieve and explained why, this part focuses on how to do it. The guiding principles developed here together with the tools of the trade introduced in the previous chapter form the foundation on which the patterns in the third part are built.

We decided to present the material in a contiguous and cohesive fashion so that it can serve as a compact reference when gauging the design of your own patterns that you will develop as you build reactive applications. As a 360 degree view of a fully reactive architecture this part covers a lot of ground. First-time readers may want to read the first of its chapters, then skim the rest of this part and return to it after having studied the corresponding patterns in the third part of this book.

In this part you will learn:

- how explicit asynchronous message passing enables encapsulation and isolation
- how location transparency improves compositionality and adds horizontal scalability
- how to structure your system in hierarchical modules following *divide et regna*[1]

Footnote 1 Latin, literal translation “divide and reign”, commonly translated as “divide and conquer”

- how this hierarchy allows principled failure handling
- how to achieve sufficiently consistent program semantics in a distributed system
- how to avoid non-determinism where possible and add it where necessary
- how to guide application design based on the topology of its message flows

# 4 Message Passing

The fundamental notion upon which message passing is built is that of an event: the fact that a certain condition has occurred (the *event*) is bundled up together with contextual information—like who did what when and where—and is signaled by the *producer* as a *message*. Interested parties will be informed by the producer using a common transport mechanism and *consume* the message.

In this chapter we will discuss in detail the main aspects around message passing:

- whether to do it synchronously or asynchronously
- the difference between focusing on messages or events
- considerations around how messages can be transmitted without overwhelming the recipient and with varying degrees of reliability

We will also see how message passing enables vertical scalability and in parting we discuss the correspondence between events and messages—or how to model one in terms of the other.

## 4.1 Synchronous vs. Asynchronous

The communication from producer to consumer can be realized in two ways:

- *Synchronous* communication means that both parties need to be ready to communicate at the same time.
- *Asynchronous* communication allows the sender to proceed without having to wait for the recipient to become ready.

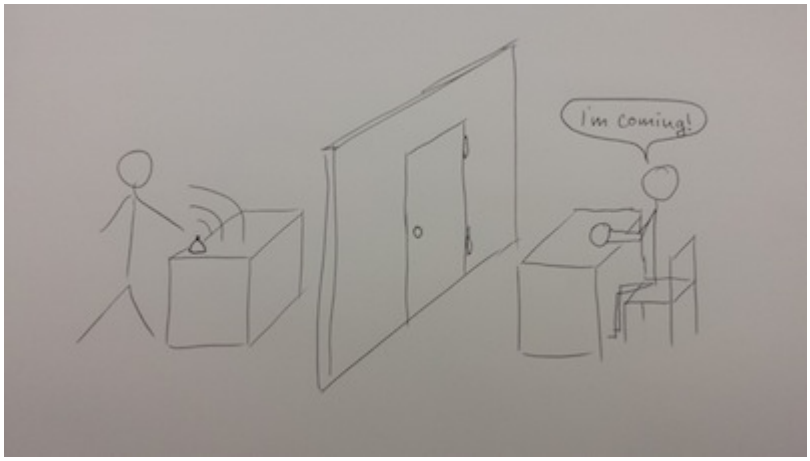


**Figure 4.1 the message that a customer is seeking assistance is passed to the clerk using a bell**

To illustrate the difference between the two approaches consider a post office as shown in figure 4.1. A customer, Jill, comes in to mail a letter and since she has run out of postage stamps she needs to ask the clerk, James, for assistance. Luckily there is no queue in front of the counter, but James is nowhere to be seen, he is probably somewhere in the back, stowing away the parcel that he just accepted from the previous customer. While she waits for him to return Jill is stuck at the counter, she cannot go to work or shop for groceries or whatever else she wants to do next. If the waiting time gets too long, Jill will give up on posting the letter for now and try again at a different post office or on the next day.

This short example demonstrates synchronous message passing: the letter is handed over from one person to the next, in this case from the sender to an intermediary who shall deliver it to the recipient. We can see that the sender is blocked from other activities while waiting for the receiver to complete the exchange and we recognize that in real life we deal gracefully with the case where the receiver is unavailable for too long—the corresponding timeouts and their proper handling are often considered only as an afterthought while programming.

In contrast, asynchronous message passing means that Jill posts the letter by placing it—with postage paid—in the letterbox. She can then immediately be on her way to her next task or appointment; the clerk will empty that box sometime later and sort the letters into their correct outboxes. This is much better for everyone involved, Jill does not need to wait for the clerk to have time for her and the clerk gets to do things in batches, which is a much more efficient use of his time. Hence we prefer this mode of operation whenever we have a choice.

One further indication in this direction arises when considering multiple recipients: it would be very inefficient to inform every single one synchronously, just as it would be very difficult to arrange and also inefficient to wait until all of them are ready to communicate at the same time. In the human metaphor the former would mean that the

producer needs to walk around, patiently waiting at every recipient's desk until they have time, while the latter would mean to arrange a full team meeting and notify everybody at the same time—but also wasting a lot of time in the process. It would be much preferable to just send an asynchronous message instead (a letter in the old times, probably an email today). Following this reasoning we will always refer to *message passing* as *asynchronous communication between a producer and any number of consumers using messages*.

**SIDEBAR** **A Personal Anecdote**

*This sidebar may be less interesting for readers who are not aware of Akka's history, but it highlights part of the learning process that inspired the author's development.*

The actor toolkit Akka has been built to express message passing from the very beginning, this is what the actor model is all about. But before version 2.0 this principle was not pervasive, it was only present at the surface, in the user-level API. Under the covers we used locks for synchronization and called synchronously into the supervisor's strategy when a supervised actor failed. An obvious consequence is that remote supervision was not possible with this architecture, in fact everything concerning remote actor interactions was a little quirky and rather difficult to implement. In addition users started noticing that the creation of an actor was executed synchronously on the initiator's thread, leading to the recommendation to refrain from performing time-consuming tasks in an actor's constructor but instead sending an initialization message.

When the list of these and similar issues grew too long we sat down and redesigned the whole internal architecture of Akka to be purely based on asynchronous message passing. Every feature which was not expressible under this rule was removed and the inner workings were made fully non-blocking. As a result all pieces of the puzzle clicked into place: with the removal of tight coupling between the individual moving parts we gained the freedom to implement the combination of supervision, location transparency (see next section) and extreme scalability at an affordable engineering price. The only downside was that certain parts—for example failure notification for the purpose of supervision as detailed in chapter 12—do not tolerate message loss, which requires more effort for the implementation of remote communication.

Turning the picture around, asynchronous message passing means that the recipient will eventually learn of a new incoming message and then consume it as soon as appropriate. There are two ways in which the recipient can be informed: it can register a

callback describing what should happen in case of a certain event, or it could receive the message in a letterbox (also called «queue») and decide on a case-by-case basis what to do with it. The difference between these two approaches is discussed in the following section.

## 4.2 Event-Based vs. Message-Based

The difference between these two models lies in which of the two is the active entity: the message or the consumer? Event-based systems focus on executing the consequences of events, typically by employing an event loop: whenever something happens the corresponding event is appended to a queue and the execution mechanism of the framework continually pulls events from this queue and executes the callbacks attached to them. The code that is executed is hereby anonymous, it is typically a small procedure that is triggered by the occurrence of a condition like a mouse click event. In reaction to an event new events might be generated which then are also appended to the queue and processed when their turn comes. This model is employed by single-threaded runtimes like node.js or by the GUI toolkits for most graphical operating systems.

In contrast, a message-based model focuses on the recipient of an event notification. The role of the anonymous callback is replaced by an active recipient that is handed messages and processes them. While an event-based system needs addressable event producers for registering callbacks, a message-based system consists of addressable consumers that can receive messages from potentially anonymous sources. Making the consumer responsible for processing its own incoming messages has several advantages:

- It allows the event processing to proceed sequentially for individual consumers, enabling stateful processing without the need for synchronization; this translates down to machine level because consumers will aggregate incoming events and process them in one go, a strategy for which current hardware is highly optimized.
- Sequential processing also means that the response to an event can depend on the current state of the consumer, previous events can have an influence on its behavior; in a callback-based scheme the consumer needs to decide up front which behavior shall be executed.
- Consumers can choose to drop events or short-circuit processing in order to respond to over-load scenarios, in short explicit queueing allows for conscious choices with regards to flow control—we will expand on that later in this chapter.
- Last but not least it matches how humans work in that we also process requests from our co-workers sequentially.

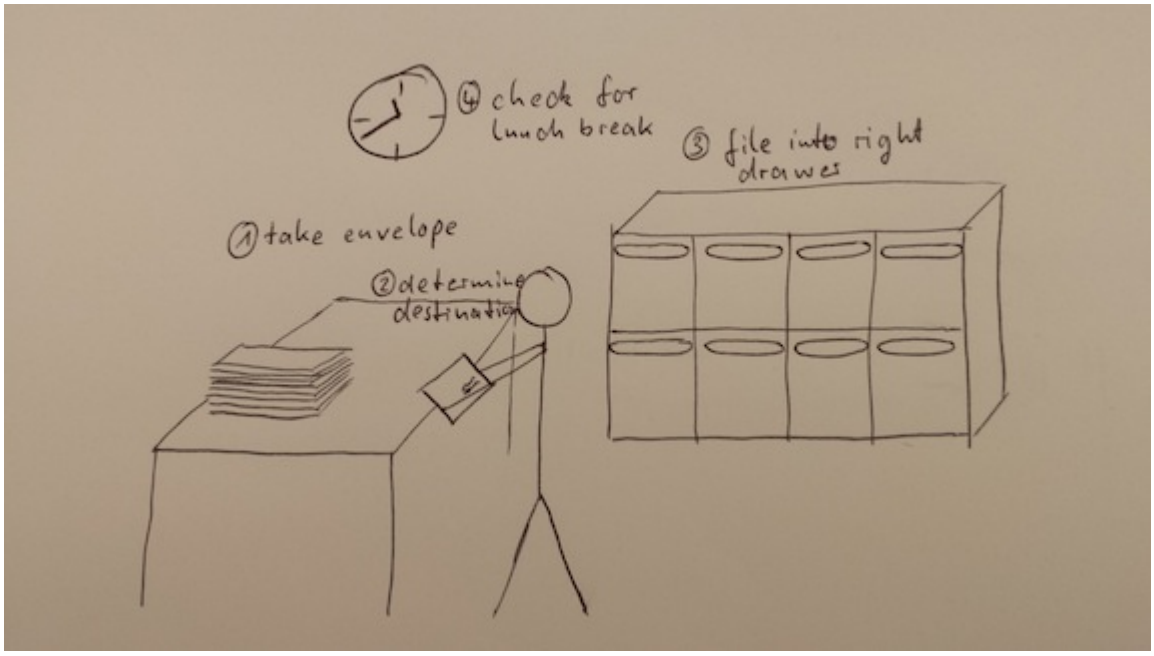


**Figure 4.2 clerk in the back-room of the post office sorts mail from incoming pile into addressee boxes**

The last point may be surprising to you but we find it helpful to visualize how a component behaves and that thrives on familiar mental images. Take a moment to imagine an old-fashioned post office as shown in figure 4.2 where the clerk sorts letters from an incoming pile into different boxes for delivery. The clerk, Beth in this case, will pick up an envelope, inspect the address label and make a decision. After throwing the letter into the right box she will turn her attention back onto the pile of unsorted mail and either pick up the next one or notice that it is already past noon and take her lunch break. With this picture in mind we already have implemented a message router, we have an intuitive understanding of it and only need to dump that into code. That task is a lot easier now than before this little mental exercise.

The similarity between message passing and human interaction goes beyond sequential processes. Instead of reading (and writing) each others' thoughts or physically reaching into each others' bodies we exchange messages, typically by speaking with each other or by observing facial expressions. This is a very important foundation upon which our civilization is built and as such it has been highly beneficial. It is therefore reasonable to express the same principle in software design by forming encapsulated objects which interact by passing messages. These objects are not the ones you know from languages like Java, C# or C++ because the communication of those is synchronous and the receiver has no say in if or when to handle the request. In the anthropomorphic view this corresponds to the case where your boss calls you on the telephone and insists that you obtain and return the answer to a question on the spot; we all know that this kind of call should be the exception to the rule lest no work gets done. What we prefer is to answer "yes, I will get back to you when I have the result", or even better the boss should have

sent an email instead of making the disruptive telephone call—especially if retrieving the wanted information may take some time. The “I will get back to you” approach corresponds to a method which returns a Future[57] while the email is equivalent to explicit message passing.

Footnote 57 a placeholder for the answer that will eventually be given, see chapter 3

Now that we have established that message passing is a useful abstraction we need to deal with two problems arising while handling messages, just like in any postal service:

1. Sometimes we must be able to guarantee the delivery of a certain very important letter.
2. If messages are sent faster than they can be delivered then they will pile up somewhere and finally either the system collapses or mail will be lost.

We will look at the issue of delivery guarantees in a minute, but before that we will take a peek at how reactive systems control the flow of messages to ensure that requests can be handled in a timely fashion and without overwhelming the receiver.

## 4.3 Flow Control

Direct method invocations (as provided by languages from the C family) by their nature include a very specific kind of flow control in that the sender of a request is blocked by the receiver until processing is finished. In case of multiple senders competing for the receiver’s resources it is commonly the case that processing is serialized through some form of synchronization—for example locks or semaphores—which means that the sender is additionally blocked until processing can commence. This form of implicit back-pressure may sound convenient in the beginning, but as systems grow and non-functional requirements become more important it turns into an impediment. Instead of implementing the business logic you will find yourself debugging performance bottlenecks.

Message passing affords you more freedom in how flow control shall behave in your application since it includes the notion of queueing in its design. As discussed in chapter two a queue can for example have a bounded size and upon reaching that size you have the option of dropping either the newest, the oldest or all messages[58], depending on the exact requirements. Another freedom is that the incoming messages can be sorted into multiple queues based on priorities and dequeued according to specific bandwidth allocations. Dropped messages can also generate an immediate negative reply if desired; we will discuss these and other possibilities in chapter 15.

Footnote 58 An example for dropping all buffered messages in case of queue overflow would be a real-time system which aims at displaying the latest data. This may include a small buffer to smooth out processing or latency jitter, but keeping old messages in the buffer is less important than getting the newest information through to the user as quickly as possible.

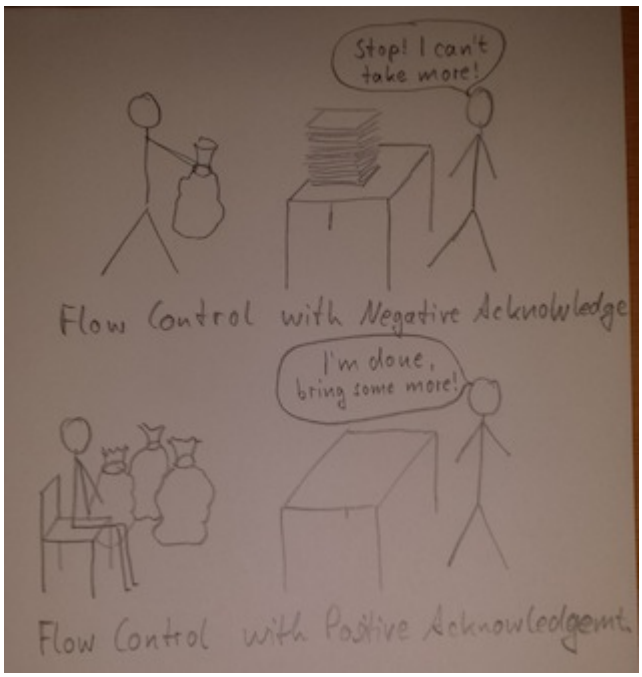


**Figure 4.3 two basic flow control schemes between post office clerks**

Two basic flow control schemes are depicted in figure 4.3: the left clerk tries to deliver more sacks full of letters to the right clerk who will sort them. Either the right clerk rejects the delivery when the table is full—which is called negative acknowledgement or *NACK*—or the left clerk waits patiently for the right clerk to inform them that he has run out of letters to sort, called positive acknowledgement or *ACK*. There are many variations on this scheme some of which are presented in chapter 15. The subject lends itself well to human metaphors like the one given here, feel free to let your mind roam and discover some of them in our civilized day to day life.

In essence message passing unties the package which comes with common object-oriented programming languages and allows customized solutions. It is clear that this choice does not come without a cost—at the least you need to think about which flow control semantics you want—hence you need to choose the granularity at which message passing is applied and below which direct method calls and object-oriented composition are preferable. As an example imagine a service-oriented architecture with asynchronous interfaces. The services themselves might be implemented in traditional synchronous style and message passing is applied as the glue between them. When refactoring one of the service implementations you might find it appropriate to apply more fine-grained flow control within it, thus lowering the granularity level; this choice can be made differently depending on the service's requirements as well as how the responsible development team likes to work.

With a basic understanding of how not to overload a message delivery system we can turn our attention to the issue of how to ensure delivery of certain important messages.

## 4.4 Delivery Guarantees

In the example of the post office it becomes clear that despite the clerk's best efforts sometimes a letter gets lost. The probability might be rather low but still it happens—several times per year. What happens in that case? The letter might have contained birthday wishes, in which case it is hopefully discovered by sender and receiver when they meet for the next time. Or it might have been an invoice which will consequently not be paid and a reminder will be sent. The important theme to note here is that humans normally follow up on interactions they have, allowing the detection of message loss. In some cases there will be unpleasant consequences after a message loss, but life goes on and we do not stop our whole society because one letter was not delivered. In the following we will show you how reactive applications work in the same way, but we will start at the opposite end—with synchronous systems.

When we write down a method call then we can be pretty certain that it will be executed when the program reaches the corresponding line, we are not used to losses between lines. But strictly speaking we would need to take into account the possibility that the process will terminate or the machine will die; it could also overflow the call stack or otherwise raise a fatal signal. It is not difficult to think of a method call as a message sent to an object[59], and with this formulation we could say that even in a synchronous system we have to factor in the possibility that a message—a method invocation—can get lost. We could thus take the extreme standpoint that there cannot be an unbreakable guarantee that a request will be processed or result in a response.

Footnote 59 In fact this is the original idea behind object-oriented design as pioneered in Smalltalk-80.

This stance would not be deemed constructive, however, because we tend to see nothing as absolute, we always accept implicit limitations and exceptions to the rule—we apply common sense and call those who don't pedantic. For example, human interaction usually proceeds under the implicit assumption that neither party dies. Instead of dwelling on the rare cases we concern ourselves more with managing our inherent unreliability, making sure that communication was received, reminding colleagues of important work items or deadlines. When transforming a process into a computer program we expect it to be completely reliable, to perform its function without fail. It is our nature to loathe the hassle of countering irregularities and we turn to machines to be rid of it. The sad fact is that the machines we build may be faster and more reliable, but they are not perfect and we must therefore continue to worry about unforeseen failures.


**Figure 4.4 the retry pattern in our daily life**

The established procedures between humans are again a good model in this regard. If one person requires another's service and sends a request then they will have to deal with the possibility that there will be no reply. It might be that the other person is busy, or it might be that the request or reply was lost along the way, for example because the letter (or email) was lost. In these cases the request needs to be retried until it is met with a response or becomes irrelevant, as shown in figure 4.4. The transfer of this mode of operation into the realm of computer programming is obvious: a message sent from one object to another may be lost, or that other object might be unable to handle the request because something it depends on is currently unavailable—disk full or database down—which means that the sending object needs to either retry until the request is successful or give up.

Handling this within the framework of synchronous method calls is of questionable utility because failure usually implies that the requesting object's behavior cannot be executed as well—for example due to abnormal program termination. In the case of message passing it is possible to persist the request and retry it once the failure condition has been corrected; the program can continue after a complete outage of a computing center as long as the needed messages were committed to stable storage. Analogous to flow control the granularity at which this is applied needs to be chosen appropriately for each application based on its requirements.

With this in mind it becomes very natural to design a system with reduced message delivery guarantees. Building this knowledge in from the beginning makes the resulting application resilient against message loss, independent of whether that is caused by a network interruption, a back-end service outage, excessive load or programming errors.

The flip-side of this coin is that implementing a runtime environment with very strong delivery guarantees is expensive in the sense that extra mechanisms need to be

built in—for example re-sending network messages until receipt is confirmed—and these mechanisms will reduce performance and scalability for all uses, even when no failures occur. The cost rises dramatically in a distributed system, mostly because confirmations require network round-trips which have a latency that is orders of magnitude larger than on a local system (for example between two cores of a single CPU package or socket). Providing weaker delivery guarantees means that the implementation of the common case can be a lot simpler and faster and you only pay the price in those cases where stronger guarantees are needed; note how this also corresponds to how the postal service distinguishes between normal and registered mail.

The principal choices are the following:

- *At-Most-Once Delivery:* This is the most basic form which has the lowest implementation overhead: a request is sent once and never retried. If it is lost on the way or processing fails then no recovery is attempted. Therefore the desired function might be invoked once or not at all. This scheme is cheap to implement because neither sender nor receiver are required to maintain information on the state of their communication.
- *At-Least-Once Delivery:* Trying to guarantee that a request is executed requires two additions to at-most-once semantics. Firstly the recipient must acknowledge receipt or execution—depending on the requirements—and secondly the sender must retain the message in order to be able to resend it as long as no confirmation is received. Since a resend can be the result of a lost confirmation this means that the recipient may receive the message more than once. Given enough time and the assumption that communication will eventually succeed this means that under this scheme the message will be received at least once.
- *Exactly-Once Delivery:* If a request must be processed, but it must not be processed twice, then in addition to at-least-once semantics the recipient must de-duplicate messages, it must keep track of which requests have already been processed. This is the most costly scheme since it requires communication state to be kept at the sender as well as the receiver side. If at the same time processing must happen in strict order then throughput may be further limited by the necessity to complete confirmation round-trips for every request before proceeding with the next—unless flow control requirements are compatible with buffering at the receiver side.

Implementations of the actor model usually provide at-most-once delivery out of the box and allows the addition of the other two layers on top for those communication paths which require it. Interestingly, local method calls also provide at-most-once semantics, albeit with a very tiny probability for non-delivery.

The discussion of flow control and message delivery guarantees is important because message passing makes limitations of communication explicit and clearly exposes inconvenient corner cases. In the next sections we focus on the benefits that message passing brings: they can be processed in parallel where appropriate and they match real-world events quite naturally.

## 4.5 Vertical Scalability

The good old post office probably had only one clerk sorting the mail because at least in my mind it was situated in a small town in the middle of nowhere. But imagine a busy post office in Manhattan back when computers and machines were not yet capable of sorting letters. You could have multiple clerks sort them in parallel, speeding up the process accordingly as shown in figure 4.5. The analog of this can be applied also to reactive applications wherever the order of requests is not crucial—we will discuss specifics in *Message Routing Patterns* in chapter 14—but we will again start out from a synchronous system to make the advantage clear.

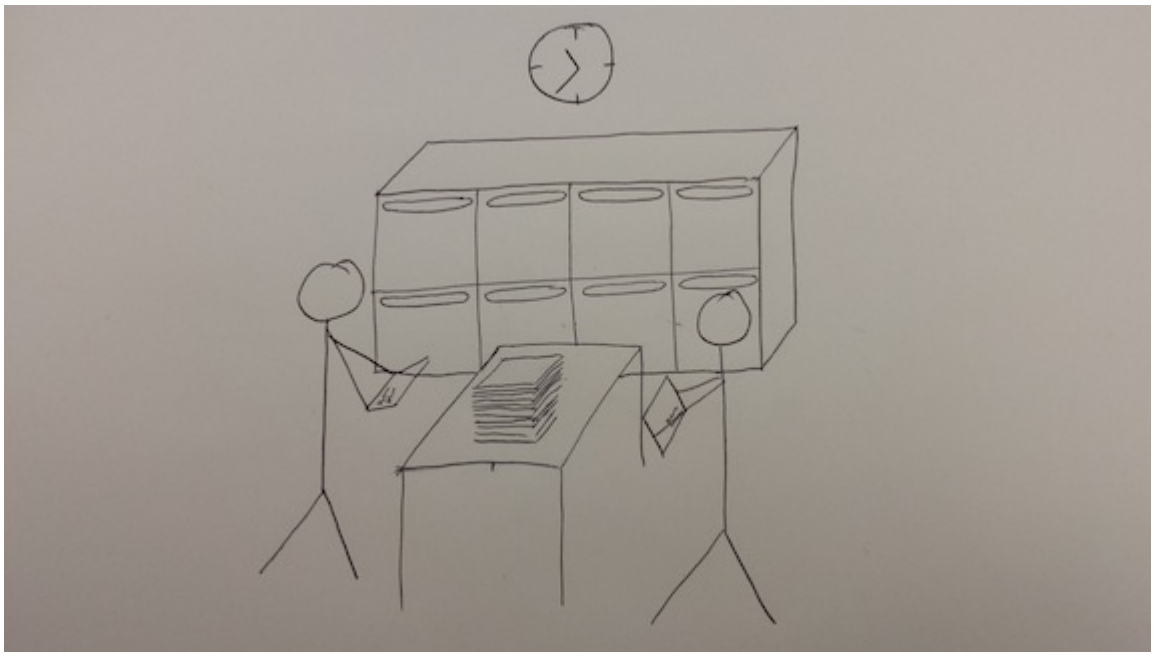

**Figure 4.5 two clerks sorting mail in parallel at the post office**

Imagine a piece of code which performs an expensive calculation (prime factorization, graph search, XML transformation or whatever you want). If this code is accessible only by synchronous method calls then the onus is on the caller to provide for parallel execution in order to benefit from multiple CPUs, for example by using Futures to dispatch the code as tasks onto a thread pool. One problem with this is that only the implementer of that code knows whether it can be run in parallel without concurrency issues or not; a typical problem arises when auxiliary data structures are used in the computation that are stored and reused within the called class or object without proper internal synchronization.

Using message passing decouples sender and receiver in a way which corrects this design error. The implementation of the calculation may choose to provide several execution units and distribute incoming messages across them without the sender needing to know about this detail. This could be realized by having multiple worker threads pull

work items from the incoming message queue, which is only possible because asynchronous execution means that the resources used for fulfilling the request may not be the caller's, they may not even be accessible to the caller and they may be more powerful than what the caller has at its disposal.

The beauty of this approach is that the semantics of message passing do not change depending on how the message will be processed. Scaling a computation vertically on dozens of processor cores can be realized transparently for the calling code, hidden as an implementation detail or a configuration option.

## 4.6 Events as Messages

The other advantage besides vertical scalability is that message passing naturally fits the way computers interact: in the end it always boils down to events being raised and handled. To recapitulate, an event is a condition which occurs at a specific point in time and it depends very much on the circumstances how urgently it needs to be handled. In hard real-time systems the foremost concern is placing stringent limits on the maximal response time to external events, expressing a tight temporal coupling between production and consumption. At the other end of the spectrum a high volume storage system for persisting log messages will favor throughput over latency, here it does not matter much how long it takes as long as the message is stored eventually.

An event which propagates through a system can also be seen as a message which is forwarded along a chain of processing units and there is a fuzzy line between real-time processing and unbounded asynchrony. Representing events as messages enables the trade-off between latency and throughput to be adjusted on a case-by-case basis or even dynamically. This is useful to model use-cases where usually response times shall be short but the system shall also degrade gracefully when the input load increases.

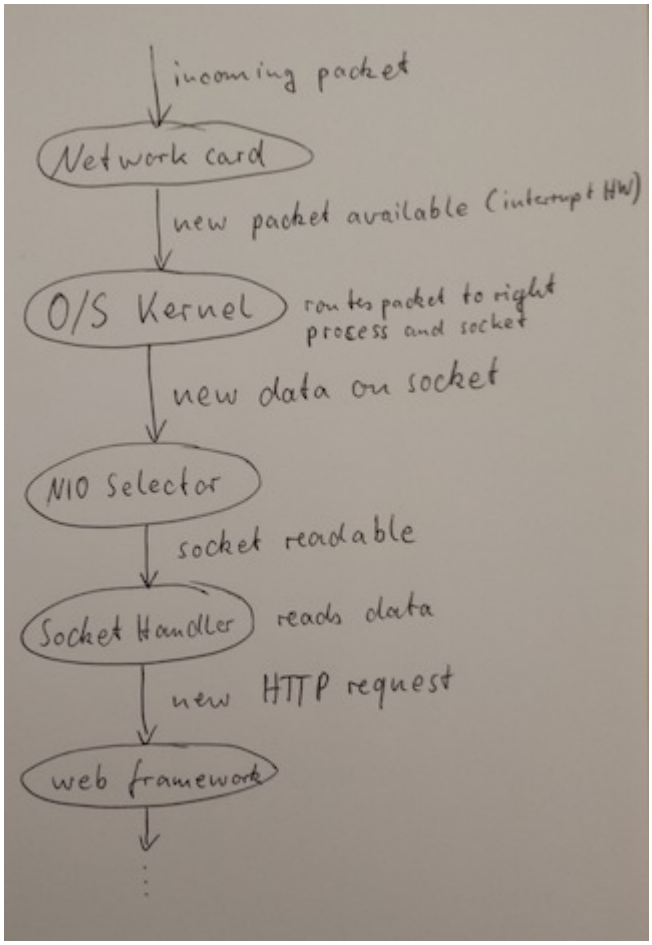


**Figure 4.6 showing the different steps of a web request from reception at the NIC to the web framework**

An example of varying processing requirements is handling the reception of network packets by a computer as shown in figure 4.6. First the network adapter informs the CPU of the availability of data using a synchronous interrupt which needs to be handled as quickly as possible. In response to this interrupt the operating system kernel takes over the data from the NIC and inspects it to find the right process and socket to transfer it to. From this point onwards latency requirements are much relaxed because the data are safe and sound within the machine's memory, but of course the user-space process which wants to reply to the incoming request benefits from being informed as quickly as possible. The availability of data on the socket is signaled—for example by waking up a selector—and the application can then request the data from the kernel.

You will notice a pattern here: the reception of data from the wire propagates as a series of events from one layer to the next. It is therefore natural to model network I/O within the application as events, reified as a stream of messages. Each successfully received packet—possibly combined with following ones for efficiency—will end up in the user-level program in a representation which matches its underlying nature. We picked this example because for Akka we recently re-implemented the network I/O layer

in this fashion, but opportunities for exploiting this correspondence are ubiquitous. Every interaction between objects in the real world ultimately proceeds through a form of message passing, at a fundamental level sound or light waves represent our most important means of communication. All inputs of a computer are event-based (keyboard and mouse, camera and audio frames) and can conveniently be passed around as messages. In this way message passing is the most natural form of communication between independent objects.

## 4.7 Synchronous Message Passing

Explicit message passing is not only useful in the communication between isolated compartments of an application, it can also be very beneficial towards delineating components in the source code. Where asynchrony is not needed for decoupling these at runtime it only presents an added cost—dispatching tasks for asynchronous execution incurs administrative runtime overhead as well as extra scheduling latency.

We mention this because synchronous message propagation is useful for stream processing for example when fusing a series of transformations together, keeping the transformed data local to a CPU core and thereby making better use of the associated caches. This kind of messaging thus serves a different purpose than what we need for compartmentalizing a reactive application and our derivation of the necessity of asynchrony in this section does not apply to the use of message passing discussed here.

## 4.8 Summary

In this chapter we have discussed in detail the motivation for using message passing, in particular in contrast to synchronous communication. We have have illuminated the difference between focusing on addressable event sources in *event-driven* systems and addressable message recipients in *message-driven* systems. As a consequence of explicitly considering message passing we have encountered the need for flow control and we have learnt about the different levels of message delivery guarantees.

We have briefly touched upon the correspondence between events and as a benefit resulting from message passing we have seen how it enables vertical scalability. In the following chapter we will add the notion of location transparency which will complement this observation by enabling horizontal scalability.

# 5 Location Transparency

The previous chapter introduced the concept of message passing as a way of decoupling collaborating objects. Making communication asynchronous and non-blocking instead of using synchronous method invocations enables the receiver to perform its work in a different execution context, for example on a different thread. But why stop there? Message passing works in the same way for local and remote interactions, there is no fundamental difference between scheduling a task to run later on the local machine and sending a network packet to a different host to trigger execution there.

In the following we will explore the possibilities offered by this perspective as well as the consequences this has for quantitative aspects like latency, throughput and probability of message loss.

## 5.1 What is Location Transparency?

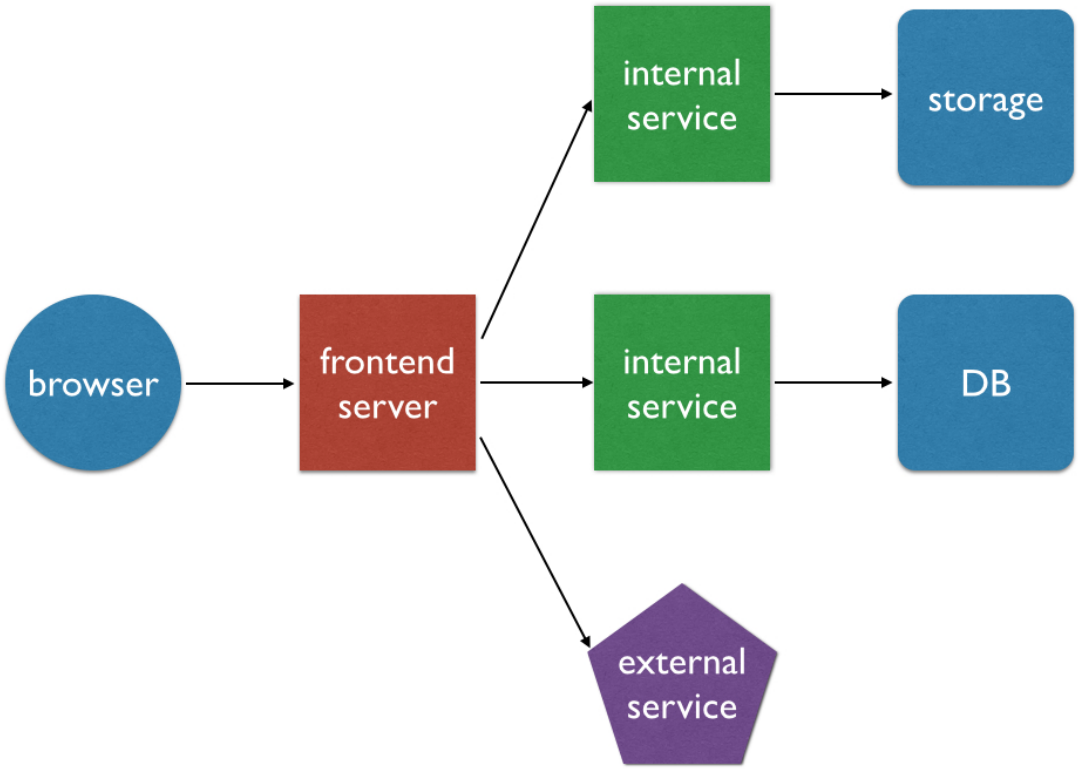


**Figure 5.1 services deployed for a typical web application**

You may recall the example shown in figure 5.1 from chapter two, where we discussed the tenets of the Reactive Manifesto using a hypothetical—and much simplified—view of the Gmail application. If you were to start designing this system you would split it up into services with various presumed dependencies between them, you would start mapping out the interfaces which support inter-service communication and you would probably spend a thought or two on how the whole application will be deployed on the available hardware (or which hardware needs to be made available). This design will then be carried out by implementing each of the services, codifying the drafted dependencies and communication patterns, for example using an asynchronous HTTP client to talk to—or exchange messages with—another service's REST API or using a message broker.

Message passing in the above example is clearly marked out within the program code, using different syntax than for local interactions. This means that the calling object needs to be aware of the location of the receiving object, at least it needs to know that the receiver does not support normal method calls.

Location transparency means that sending a message always looks the same, no matter where the recipient will process it. Application components interact with each other in a uniform fashion based on explicit message passing. The object which allows sending a message then becomes just a handle pointing to its designated recipient, it is mobile and can be passed around freely among network nodes.

## 5.2 The Fallacy of Transparent Remoting

Since the advent of ubiquitous computer networks several attempts have been made to unify the programming model for local and remote method invocations: CORBA, Java RMI, DCOM were all introduced with this promise and all of them failed. The reason for this lies in a mismatch of implied and actual guarantees or execution semantics as for example Waldo et al[60] have noted.

Footnote 60 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

The most obvious problem is that of partial failure. When calling a method you expect either a result or an exception, but if that method call is routed to a different computer via a network then there are more things that can go wrong. The method could not be called at all because the invocation cannot be transmitted, or the method is invoked and the result is lost; in both cases there would be no outcome to report, which is typically solved by raising a timeout exception in the caller's thread. This means that the calling code must deal with the uncertainty of whether the desired function was invoked or not.

Other problems arise from the performance expectations associated with a method call. Traversing the network incurs an increase in latency of at least one round-trip time—the invocation needs to be sent to the recipient and the result needs to travel back to the sender—and this increase is several orders of magnitude larger than the overhead of executing a method locally. While the latter takes about a nanosecond on current hardware the former is measured in (tens to hundreds of) microseconds in a local network and up to hundreds of milliseconds for globally distributed systems. The implementation of an algorithm needs to take this latency into account in order to fulfill its non-functional requirements, and if an innocent looking method call in fact takes a million times longer than one would naïvely expect then that can have devastating consequences.

In addition to increased latency remote invocations also suffer from much lower throughput than their local counterparts. Passing huge in-memory datasets as arguments to a local method may be slow due to the need to bring the relevant data into the CPU caches, but that is again much faster than serializing the data and sending them over the network. The difference in throughput can easily exceed a factor of 1000.

Therefore the conclusion is that offering remote interactions with the same semantics as local ones is neither desired nor has it been successful. Waldo et al argue that the

inverse—changing local semantics to match the constraints of remote communication—would be too invasive and should therefore also not be done; they recommend clearly delineating local and remote computing and choosing different representations in the source code.

## 5.3 Explicit Message Passing to the Rescue

We have seen in the previous section that message passing models remote communication semantics quite naturally: there is no preconceived expectation that messages are transmitted instantaneously or processed immediately, replies are also messages and messages may even be lost. Two components communicating with messages can thus be located on the same machine or on different network hosts without changing any characteristic quality of their interaction. The inner workings of these components can comprise any number of objects collaborating by synchronous local method calls and it is important to note that message passing needs to be explicitly different as argued in the previous section.

Location transparency does not aim at making remote interactions look like local ones, its goal is rather to unify the expression of message passing under a common abstraction for both local and remote interaction. A concrete example of this is the Actor Model as shown in chapter 3 where all communication between actors is mediated through stable place-holders—called *ActorRef* in Akka or *Process ID* and *Registered Name* in Erlang—whose only purpose is to offer a generic message passing facility.

```
actorRef ! message
```

The code snippet will send the message given on the right to the actor represented by the reference on the left. Note how this operation does not involve calling a method on the actor itself, instead the actor reference is a façade which will transfer the message—if necessary across the network—to where the actor is located and append it to its incoming message queue for later processing.

Contrast this with the traditional way of *transparent remoting* which would involve creation of a local proxy object and calling the desired method directly on it, thereby hiding the fact that message passing is used underneath[61]. This is the crucial point where transparent remoting and location transparency differ.

---

Footnote 61 Additional concrete problems arise in defining the semantics of methods that do not return a value: shall the current thread be blocked until the remote object has processed the call, or shall it return to the caller immediately?

---

Another way to look at it is that by treating message passing always as potentially remote we gain the freedom to relocate components without having to change their code, we can write software once and deploy it on different hardware setups by just changing

where the parts are instantiated. Akka for example allows remote deployment of actors to be specified in the configuration file, which means that the operations team can in principle determine the hardware layout without regards for which partitioning the engineering team foresaw. Of course this depends on the initial assumption that all message passing was implemented as if it was remote to begin with, and it would be prudent to consult between development and operations teams in order to have a common understanding of the deployment scenarios to optimize for.

Local message passing—while looking the same in the source code—then becomes merely an optimization of remote message passing. This could mean that messages are passed by reference instead of serializing and deserializing them, greatly reducing latency and increasing throughput compared to the remote case. Erlang takes the correspondence between local and remote one step further in that it does not apply this optimization and serializes and deserializes all message sends, including purely local ones; this of course increases the decoupling of different components from each other at the cost of performance.

## 5.4 The Role of Latency and Throughput

Writing location transparent code does not stop with using a toolkit which supports this paradigm, we have just seen that local optimizations can make message passing a lot faster than remote messaging. If the implementation of an algorithm depends on this optimized form of message passing then it will break or perform poorly when deployed across multiple network hosts.

The advantages or message passing as discussed in the previous section—loose coupling, horizontal scalability, flow control—can of course be desirable for a purely local program, and in this scenario it does not make sense go to great lengths in minimizing the serialized size of transmitted messages. Therefore location transparency is a feature that you can use to great effect on a higher level of abstraction while choosing to not apply it within sub-programs. A possibility is to apply it at the service level in a service-oriented architecture and lowering the granularity into the implementation of a service if its deployment will likely not fit onto a single machine. We will discuss this question further in the following sections.

## 5.5 The Role of Message Loss

We have seen above that remote communication has different latency and throughput profiles from local in-process communication. Another difference is that the probability for message loss is higher than on the local machine since more parts are involved: network hardware may be faulty or overloaded, leading to packet loss, and while using TCP mitigates the effect for certain kinds of errors it is not a panacea—connections can be dropped by any of the active network components in which case TCP is powerless.

A message being lost is not the only possible explanation in case no reply is received:

it might also be that processing failed in an unforeseen way such that no message was sent in return. You might be tempted to just apply catch-all error handlers which send back a failure message, but this does not ensure that a reply will be generated either. System resources might be exhausted (it might run out of memory or the receiving object does not get scheduled on a CPU for a long time) meaning that the reason for failing to process a message may be out of reach for the code which is supposed to be executed.

When a message is lost or not processed the only information the sender gets is that no confirmation comes back. Taking a step back we can distinguish two cases: either the sender was not interested in a confirmation because the operation was not that important (debug logging for example does not require confirmation) or it needs to react to the reception or absence of the confirmation.

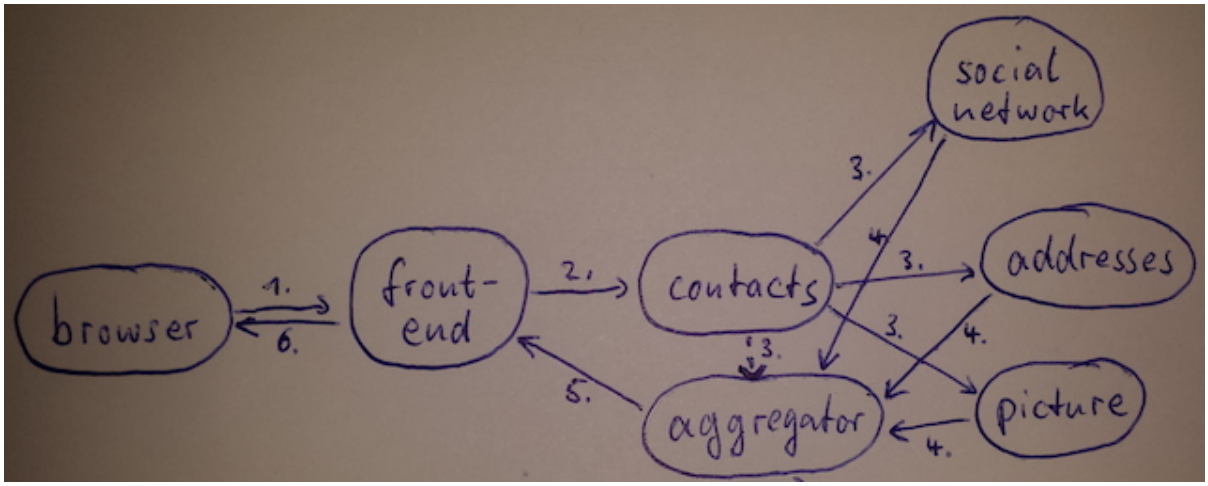


**Figure 5.2 request and reply chain when retrieving contact information**

To illustrate the second case we return to the example of the Gmail application. When you hover your pointer over the sender of an email a small contact card will pop up, showing that person's connection to you in Google's social network, a picture, possibly alternative email addresses, and so on. In order to display this window the web browser will have to make a request as shown in figure 5.2 to the Gmail front-end servers which will then dispatch requests to the different internal services that contain the desired information. The front-end service will store contextual information about where the result shall be sent to. Replies from the internal services are added to this context as they come in and when the information is complete the aggregated reply is sent back to the original requester. But what shall happen if one of the expected replies does not arrive?

Absence of a message can be expressed by triggering a timeout of the kind "send me this reminder in 100 milliseconds", then the next step depends on which message is received first. If the desired reply arrives before the timeout then processing continues normally and the timeout can be canceled if the system supports that. If on the other hand the timeout arrives before any suitable reply then the recovery path needs to be executed,

possibly resending the original message, falling back to a backup system or default value, or aborting the original request. This is depicted in figure 5.3.

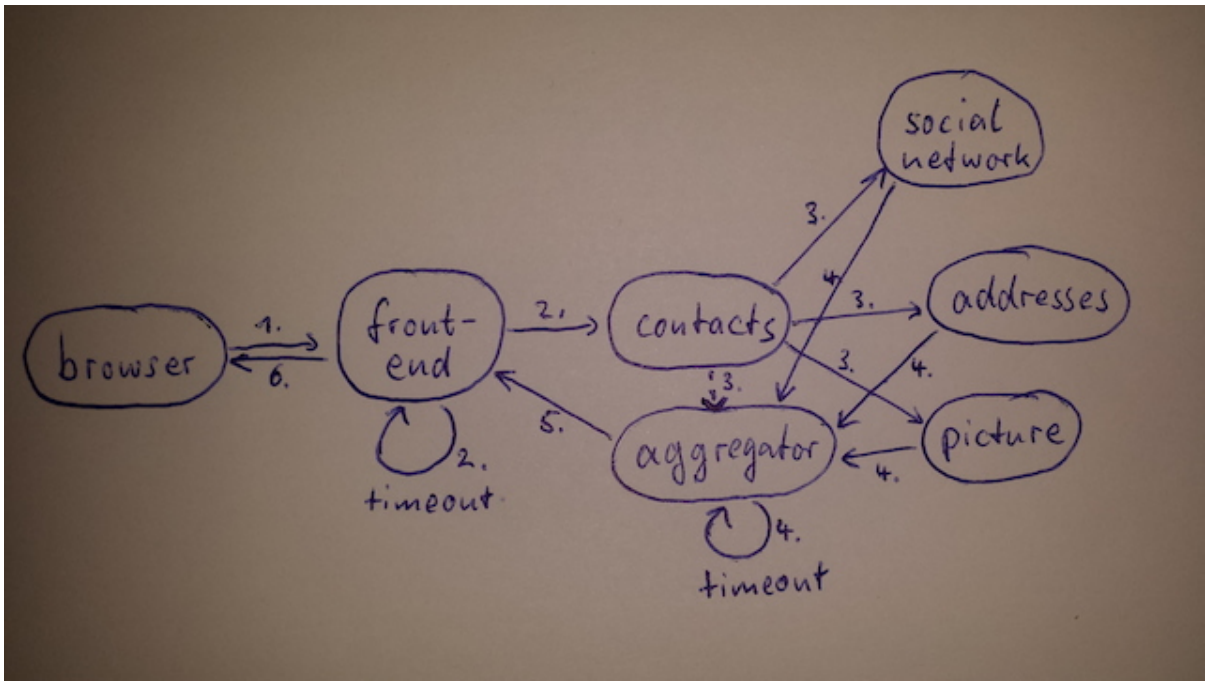


**Figure 5.3 request and reply chain with timeout**

In the contact information pop-up example there are different possibilities depending on which reply is missing. As long as the social network status can be retrieved it makes sense to send a partially successful reply, for example missing the alternate email addresses part. But the request will have to be answered with an overall failure if the social network is unavailable. Patterns like this are described in *Result Aggregation Patterns* in chapter 14.

As we discussed in the second chapter under the topic of bounded latency, timeouts have the disadvantage that they need to be set wide enough to allow for high-latency responses in order not to trigger too often. When a back-end service stops responding it makes sense to fail requests early and only send out a message every few seconds to check whether it has come back up again. This pattern is called the *Circuit Breaker* and discussed in chapter 12.

Using these patterns for your message passing applications may seem superfluous for local interactions and you might think that a local optimization would be to assume successful delivery of all messages, but as shown above processing faults can contribute to the same failure symptoms. As a result of writing your application under the assumption of remote communication—and by applying location transparency—you make it more resilient at the same time. Like with message passing the benefit results from being explicit about the recovery mechanisms in case of failure instead of relying on implicit but incomplete hidden guarantees.

## 5.6 Horizontal Scalability

In the previous section we have seen that message passing decouples caller and callee, turning them into sender and receiver. This enables vertical scalability because the receiver is now free to use different processing resources than those of the sender, a feat which is made possible by not executing both on the same call stack.

Combined with location transparency this allows the placement of the receiver anywhere on a reachable computer network. For example in Akka an ActorRef could refer to a single actor on the local node, but it could also dispatch the messages sent through it to a set of actors spread out over a compute grid. The method which is called for sending messages is the same in both cases, the difference is not visible at the use site.

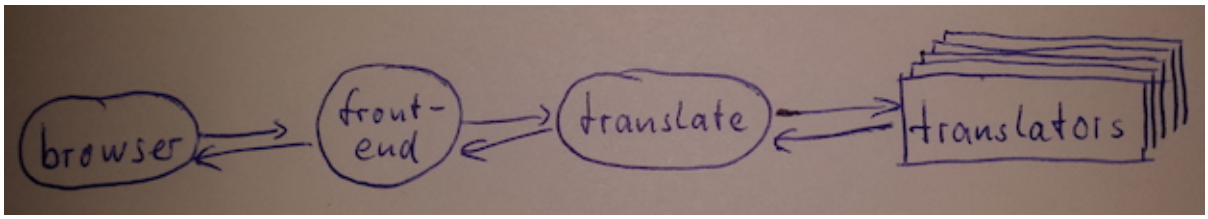


**Figure 5.4 scaling out to multiple translation services**

In the example of the Gmail application consider the translation service: normally you see the email as the author sent it, but if the application determines the language to be foreign to you it will show a link that allows you to have it translated. The service which performs this expensive transformation can be replicated across a set of computers as many times as needed to fulfill its throughput requirements. Location transparency allows this to happen transparently as shown in figure 5.4, which enables this to be done on an as-needed basis: developers typically want to test it on their notebook, the staging test-bed should contain several remote nodes for its deployment, and then in production the operations team is free to scale it up and down as far as they need to react to load spikes.

This flexibility is obtained through location transparent message passing. Further discussion can be found in *Message Routing Patterns* in chapter 14.

## 5.7 Location Transparency makes Testing Simpler

The previous point deserves separate discussion from a different perspective. Horizontal scalability does not only work outwards, it also allows running the whole system on a single computer when desirable. One use-case is during functional testing and local exploration during development, but also continuous integration testing becomes a whole lot simpler if the service under test can be wired to local communication partners or even stubs of them. You have probably written similar code which connects to an ephemeral local database for the purpose of testing, using for example classical dependency injection by configuring the database's URI.

Depending on the concrete implementation stubbing out a service can be done without involving mocking frameworks or writing a lot of boilerplate code. For example Akka comes with a TestKit which contains a generic TestProbe for mocking an ActorRef. Since there is only one method to simulate—sending a message—this is all you need to replace a back-end service with a stub that can reply with success or failure according to the test plan.

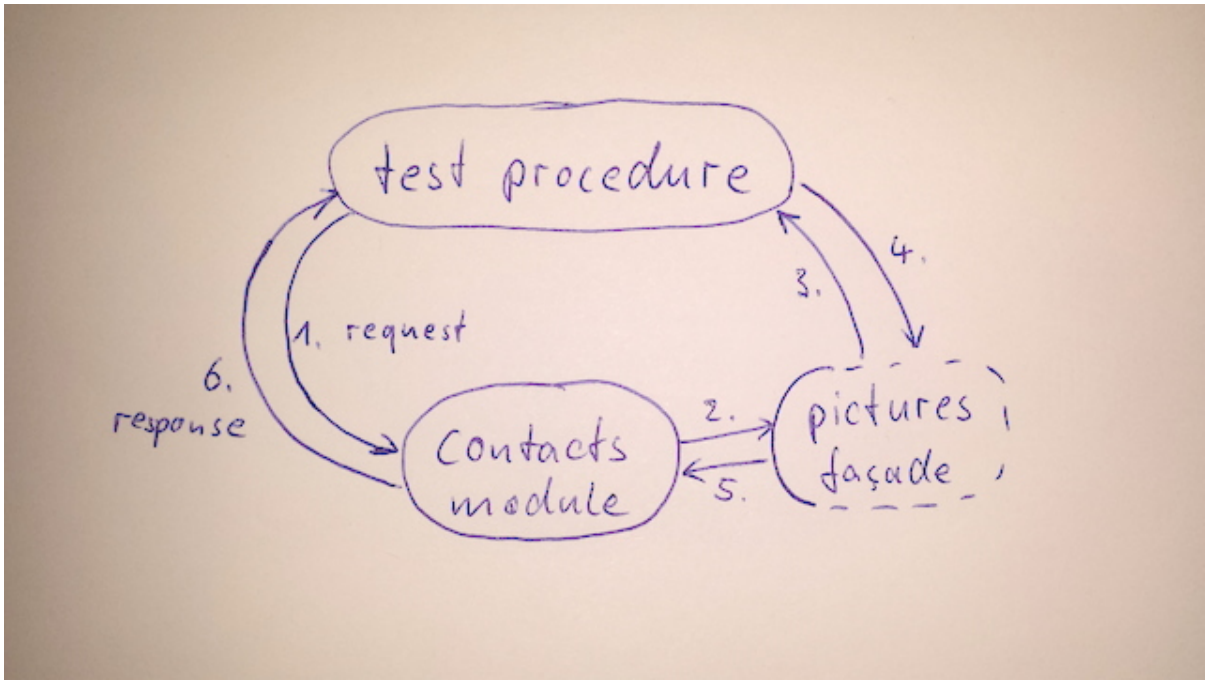


**Figure 5.5 stubbing out a service means intercepting the requests made by the module under test and injecting the replies for verification of the overall response.**

In our Gmail example the front-end services talk to several other services in order to perform their function. In order to test them in isolation we can replace for example the storage back-end for people's contact images by one which knows about a certain John Doe and which can be configured not to reply or to reply with a failure when required. An exemplary test setup is shown in figure 5.5.

This aspect is discussed in depth in chapter 11, "Testing Reactive Systems".

## 5.8 Dynamic Composition

In a service-oriented architecture wiring is usually done using a dependency injection framework which provides the exact location of each dependency. A location typically is a combination of a protocol, an address (for example host name and port) and some more protocol-specific details like a path name. Dependency resolution first determines the location for a resource and then creates a proxy object for representing it in the context into which it is injected. This process is performed during start-up and the wiring usually stays the same throughout the life-cycle of the application or service.

The second aspect of location transparency as introduced above is that handles through which messages are mediated can be sent across a network because they are in the end nothing but addresses or descriptors for the objects they refer to. In this sense they are comparable to the proxy objects created by dependency injection frameworks, but a location transparent handle can be sent across a network unlike for example a database connection handle. This enables another form of dependency injection in which one service can include references to other services in a request or reply so that the receiver uses them as a dynamic wiring.

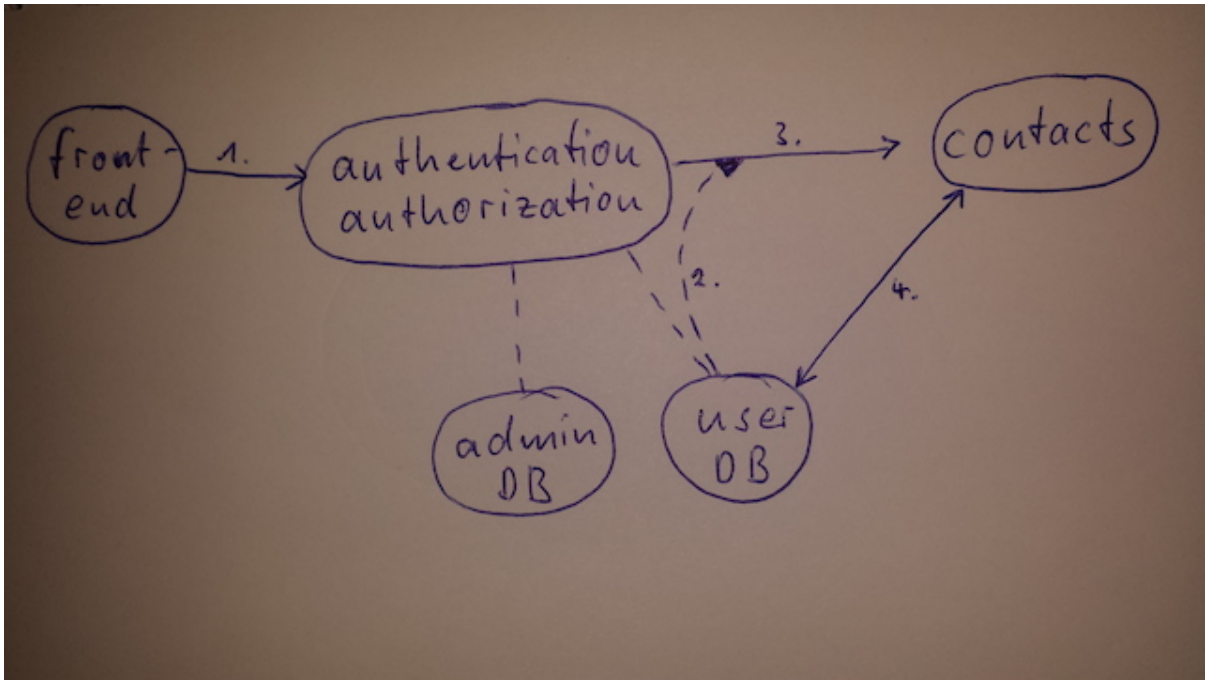


**Figure 5.6 dynamic dependency injection allows the authentication module to give access to either the user or admin database to the contacts module.**

In the exemplary Gmail application this technique could be used by the authentication service. When a request comes in the front-end will dispatch the query via this service in order to authenticate the requester and to retrieve authorization information. In addition verifying the user's access token the authentication service can inject references to other services into the request before handing it on to the contacts service as shown in figure

5.6. In this example authorization to use either the admin or the user database is passed along in the form of a service reference which the contacts module will use while processing the user request. This scheme could also be used to offer different storage back-ends or translation services with different capabilities for different subscription levels. The overall service mix available to a user can be customized in this way without having to introduce all parts everywhere or having to carry around authorization information throughout the whole system.

This concept is not new in the context of actor systems, where the capability to access a certain service is modeled by the possession of a reference to the actor providing it. Handing out this capability means introducing the reference to another actor, which will then be able to use the service on its own.

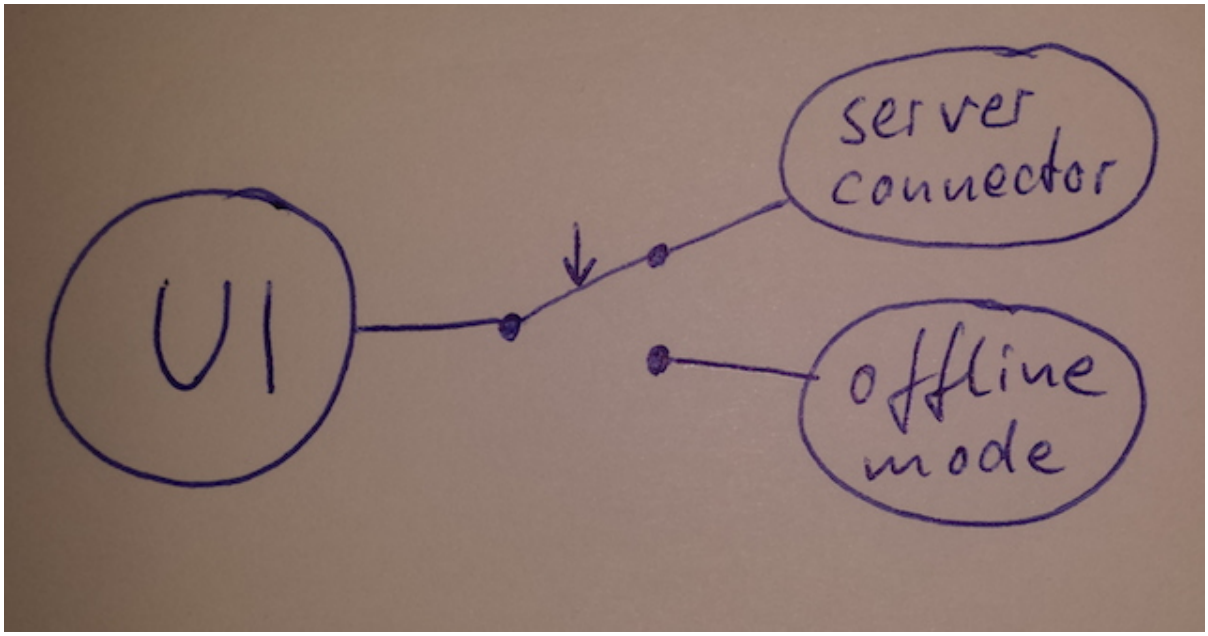


**Figure 5.7 in a web client the UI could switch between online and offline mode just by using different service references.**

Another use of this technique is to fall back to secondary services when the primary ones are unavailable, rewiring the dependency at runtime. This is useful not only on the server side: imagine a client to the blog management service which runs on a mobile device. The user interface will need to communicate with the back-end to retrieve the data to display and to persist the changes the user wants to make. When no connection is available the UI could instead talk to local stubs of the various services, for example serving cached data or queueing actions for later transmission where appropriate. Location transparency means that the UI code will not need to be aware of this difference in any way, the whole fallback logic can be confined to one service which connects the UI to either the real or a fake back-end.

## 5.9 Summary

In this chapter we have seen how explicit message passing enables us to view local messages as a special case of fully distributed remote messaging—in contrast to transparent remoting which aims at making remote interactions look like local ones. We have discussed how differences in latency, throughput and message loss probability affect our ability to unify local and remote interactions.

We have also seen the benefits of location transparent messaging, namely that it extends the vertical scalability afforded by message passing along the horizontal axis, that it eases the testing of our software components and that it allows for their dynamic composition. In the next chapter we will spend more thought on what a component is and how we break up a larger task into independent smaller ones.

# 6 Divide and Conquer

The previous section already presumed that the programs we write will typically consist of multiple parts that are segregated in some way: different areas of functionality might be developed by separate teams, modules are accessed via interfaces and packaged such that they can be replaced and so on. Over the past decades much effort has been spent on developing solutions for expressing modules within programming languages or using libraries and supporting deployment infrastructure, and we will not concern ourselves with these implementation details at this point. The more important question is how we go about dividing up a problem in order to successfully solve it? The answer to this question will be crucial for the following sections as we will see.

Imagine that we are to build the Gmail application from scratch. We begin with the diffuse intuition that the task is rather big, an entangled mess of ad hoc requirements that we come up with. One way to proceed would be to start from experience or educated guesses about which components will play a role while working on this project and postulate certain modules; there will be need for bulk storage, structured storage and relations, for authentication and authorization, for rendering and notifications, for monitoring and analytics. We will explore these, perhaps build proofs of concept, improving our understanding and letting these modules—and their siblings whose necessity we discover—gradually take shape. Bite by bite we dig into the big task, sorting it into the boxes that our modules provide. There will be small things all over the place that end up in “utils” or “misc” packages, and in the end we will have hundreds of components which will hopefully work together peacefully to allow users to read their email.

While possibly not far from the reality of many projects, the above is not ideal: by breaking down the problem into a lot of smaller problems we incur the overhead of making the individual solutions coexist in the final product. This would mean that we turned the initial frighteningly complex problem into an army of smaller problems that might overwhelm us.

Rewinding the clock by more than 2,000 years we find one of the earliest practitioners of the governance maxim *divide et regna*, Julius Caesar. The idea is quite simple: when faced with a number of enemies, create discord and divide them. This will allow you to vanquish them one by one, even though they would easily have defeated you if they had stood united. This strategy was used by the Roman empire both internally and externally, where the key was to purposefully differentiate between opponents, handing out favors and punishment asymmetrically; and it was probably[62] applied hierarchically, with senators and prefects learning from Caesar's success.

Footnote 62 The authors have no direct evidence for this, but it sounds implausible to propose the opposite.

We do not have the problem of leading a huge empire, but we can still learn from this old Latin saying: we need to break our programming problem up into a number of sub-problems that we can manage—a handful or two—and then we apply this same process hierarchically. The advantage of this approach is that we gradually narrow the scope of what we are working on, drilling down ever deeper in the details of how each particular piece will be implemented.

## 6.1 Hierarchical Problem Decomposition

Staying with the example of building Gmail, we might split the responsibilities at the top layer into sign-on, profile, contacts and mail. Within the contacts module we will have to maintain the list of contacts for every user, providing facilities for addition, removal and editing. There will also be need for query facilities, for example to support auto-completion while typing in the recipient addresses of an email (low latency but possibly imprecise) or for refined searches (higher latency by fully up-to-date and complete). The low-latency search function will need to cache a specifically optimized view on the contacts data, which will need to be refreshed from the master list when that one changes; the cache and the refreshing service could be separate modules that communicate with one another.

The important difference between this design and the initial one is that we do not only break up the overall task into manageable units—we will probably arrive at the same granularity in the end—but we also define a hierarchy among the modules. At the bottom of the hierarchy we find the nitty gritty implementation details and while moving upwards the components become more and more abstract, approaching the logical high-level functionality that we aim to implement. The relation between a module and its direct descendants is tighter than just a dependency: the low-latency search function for example will clearly depend on the cache and the refreshing service for its function, but it will also provide the scope within these two modules work—it will define the boundaries of the problem space that is solved by them.

This allows us to focus on a specific problem and solve it well, instead of trying to generalize prematurely, a practice that is often futile or even detractive. Of course

generalization will eventually occur—more likely at the lower levels of the hierarchy—but it is a natural process of recognizing that the same (or a very similar) problem has already been solved and therefore the previous solution can be reused. The higher up in the hierarchy a module is situated the more likely is it to be specific to the concrete use-case, and it is unlikely that e.g. the “mail” module of Gmail will be reused as is in a different application. If we need “mail” in another context, we are probably better off by just copying and adapting—most of the descendant modules will probably be reusable because they provide more narrowly scoped parts of the solution.

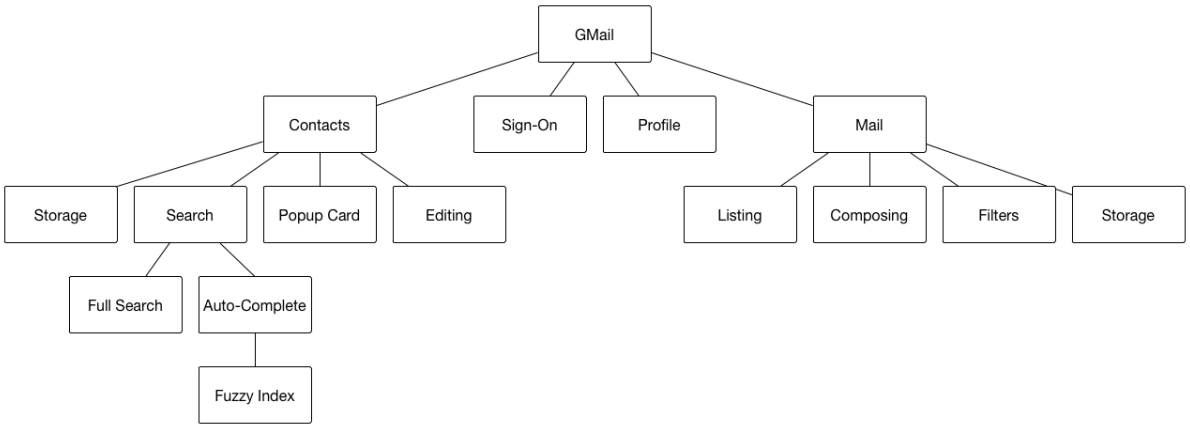


**Figure 6.1 partially decomposed module hierarchy of our hypothetical Gmail implementation**

To recapitulate: we break up the task of “building Gmail” into a set of high-level features, including “mail”, “contacts” module. The modules collaborate on providing the “Gmail” function, and that is also their only purpose. The “contacts” module provides a clearly scoped part of the functionality, and it does so by splitting that up across lower level modules including “low-latency search” which again collaborate to provide the overall “contacts” function. This hierarchy is depicted in figure 6.1.

## *6.2 Dependencies vs. Descendant Modules*

In the hierarchical decomposition process we glossed over one aspect that deserves dedicated elaboration: since the “mail” component in our Gmail example must work with contact information, does its module hierarchy contain that functionality? What we have described so far talked only about the piecewise narrowing of problem scope, and if we apply that as is then we will notice that the “contacts” functionality—or parts of it—will appear in several places: displaying lists of emails will need to have access to contact details should the user want to see them, and composing an email will need access to at least the low-latency search, the same goes for the filter rule editor.

It does not make sense to replicate the same functionality in multiple places, we would like to solve problems only once—in source code as well as in the application deployment and operations. Another factor is that neither of the above-mentioned

modules *own* the “contacts” functionality, it is not a core concern for any of them. *Ownership* is a very important notion when it comes to decomposition, in this case the question is who owns which part of the problem space. There can be only one module that is responsible for any given functionality—with possibly multiple concrete implementations—and all other modules which need to access these functions will have a *dependency* on it.

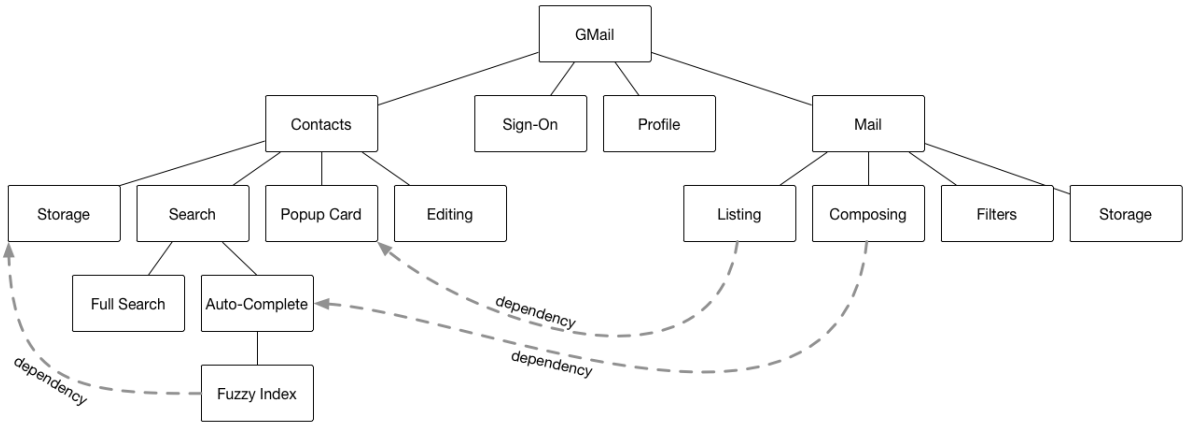


**Figure 6.2 the partial module hierarchy include some exemplary inter-module dependencies**

We hinted at this distinction earlier when saying that the relationship between a module and its descendants is tighter than just a dependency. Descendant modules cater to the bounded area of the problem space that is owned by their parent, each owning its smaller piece in turn. Functionality that lies outside of this bound is incorporated by reference and some other module is responsible for providing it as shown in figure 6.2.

## 6.3 Advantages for Specification and Test

The process of breaking down a complex task as sketched in the previous paragraphs is an iterative one, both in the sense of working your way from the root to the leaves as well as gradually refining your ability to anticipate the decision process. If we find that a component is difficult to specify—meaning that its function requires very complex description or is too vague—then we need to step back and possibly retrace our steps to correct a mistake that was made earlier on. The guiding principle is that the responsibilities of every module are very clear and a simple measure is how concise its complete specification is.

Providing a distinct scope and a small rule set to follow also benefits the verification of a module’s implementation: a meaningful test can only be written for properties that are definitive and clearly described. In this sense the acronym TDD[63] should instead of its usual expansion be taken to mean *Testability-Driven Design*; the focus on testing should not only indirectly lead to better design, problem deconstruction according to *divide et regna* should focus directly on producing modules that are easily verifiable.

Footnote 63 Normally an abbreviation for Test-Driven Development.

In addition to helping with the design process, recursive division of responsibility has the benefit of concentrating the communication between different parts of the application into pieces that can be replaced as a whole for the purpose of testing. When testing (parts of) the “mail” component of our hypothetical Gmail service, the internal structure of the “contacts” will not be of interest, the module can be stubbed out as a whole. When testing the “contacts” we can apply the same technique to its sub-modules, which includes stubbing out both the low-latency and the refined search modules. The hierarchical structure enables tests to focus only on specific levels of granularity or abstraction, replacing siblings, descendants and ancestors in the hierarchy by test stubs.

This is different from wiring a database handle into the application in either test, staging or production mode: the database handle itself is only an implementation detail of some module which uses the database for storage, let us call it “UserModule”. Only when testing the UserModule will you need to select a database server from your testbed, for all other parts of the implementation you will create an implementation of UserModule that does not use a database at all and contains test data directly. This allows everyone who does not develop the UserModule itself to write and execute tests on their personal computer without having to install the complete testbed.

## 6.4 Build your own Big Corporation

One metaphor that often works well for picturing and speaking about a hierarchical problem decomposition is that of a big corporation, for instance called *Big Corp*. At the top there is the management that defines the overall goal and direction (CEO, Chief Architect, etc.), then there are departments solving different very high-level parts of the problem, each structured into various groups of different granularity until at the bottom we reach the small teams that carry out very specific and narrowly scoped tasks. The ideas behind this structure are very similar to the problem decomposition described previously: without proper segregation of the responsibilities of departments their members would constantly step on each others’ toes while doing their work.

If you think now “well, that all sounds nice in theory, but my own corporate experience has been exactly how this does *not* work”, the good news is that while applying these techniques to a programming problem you get to decide on the structure and the relationships between parts of the hierarchy you create—you get the chance to create your own Big Corp. and do it right![64]

Footnote 64 You will probably discover that things are not as easy as you think they should be, which is equally instructive.

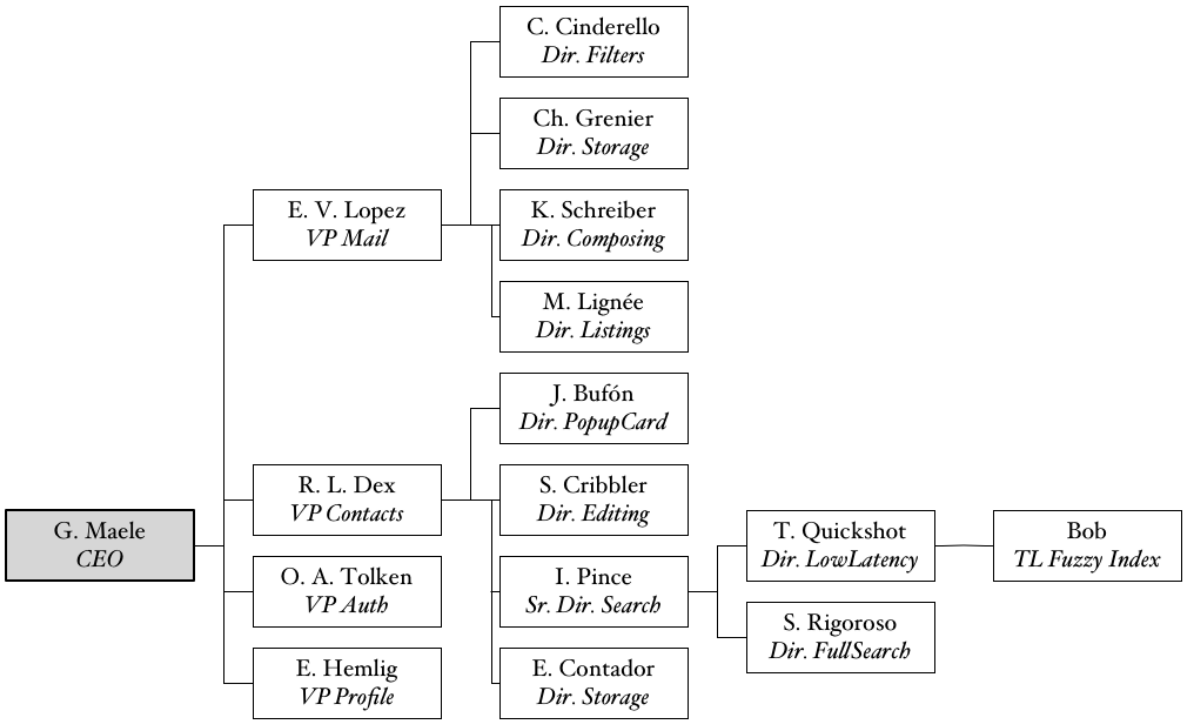


**Figure 6.3 the BigCorp view of our hypothetical Gmail application; the names are purely fictional and similarities to real persons are entirely undesired.**

The result of this view on a module hierarchy is demonstrated in figure 6.3 on the example of our venerable Gmail application. Naming the modules according to the role they play within the overall organization of the application helps establishing a vocabulary among the teams and stakeholders involved in the development.

## 6.5 Horizontal and Vertical Scalability

What have we achieved so far? If we combine the properties discussed within this chapter then we obtain modules with clearly segregated responsibilities and simple, well-specified interaction protocols that lend themselves to communication via pure message passing, regardless of whether collaborating modules are executed within the same (virtual) machine or on different network hosts. The ability to test in isolation is at the same time testament to the distributability of the components.

**NOTE** Location-transparent message passing between components that have well-segregated responsibilities enables Scalability in a way that matches the function of the application.

In the Gmail example we have encapsulated the low-latency search module in a way that makes it possible to run any number of replicas of it, which means that we are free to scale it up or down to meet the requirements of handling the load that users of the application generate. In case more instances are needed, then they will be deployed to the

available computing infrastructure, start populating their special-purpose caches and when ready they will be used to service auto-complete requests from the users. This works by having a request router set up as part of the overall low-latency search service, and whenever a new instance comes up it will register itself with this router. Since all requests are independent of each other, it does not matter which instance performs the job as long as the result is returned within the allotted time.

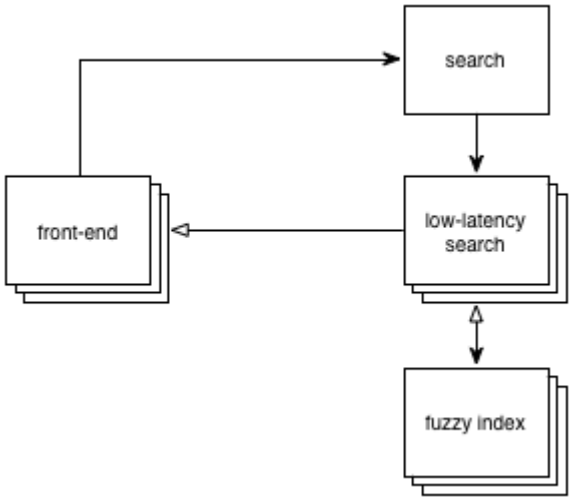


**Figure 6.4 scalable deployment of the low-latency search service, which is monitored and scaled up/down by the search supervisor**

In this example, depicted in fig X, the size of the deployment directly translates into the observed latency for the end-users. Having one instance of the search module running for every user will minimize the latency to the lowest possible value, given by network transfer times and the search algorithm itself. Reducing the number of instances will eventually lead to congestion and add latency by queueing in front of the execution of the search—in the router or on the worker instances—which will in turn reduce the frequency of search requests generated per user (assuming that the client-side code refrains from sending a new request while the old one is not yet answered or timed out).

The deployment size can thus be chosen such that the search latency[65] is as good as it needs to be, but no better. This is important because running one instance per user is obviously ridiculously wasteful, and reducing the number of instances as far as we can is in our best business interest.

Footnote 65 Remember that this will need to be formulated in terms of e.g. the 99th percentile, not the average, as detailed in chapter 2.

## 6.6 Summary

By taking inspiration from ancient Roman emperors we have derived a method of splitting up an enormous task into a handful of smaller ones, repeating this process with each one until we are left with the components that collaboratively make up our whole application. Focusing on responsibility boundaries allowed us to distinguish between who uses another component versus who owns it. We have illustrated this procedure on the example of how a big corporation is structured.

On this journey we have seen that such a hierarchical component structure benefits the specification and thereby the testing of our components and we have also noted that components of segregated responsibilities naturally are units for scaling our application deployment vertically as well as horizontally. Another benefit of this structure will be studied in the next chapter when we talk about failure handling.

# 7 Principled Failure Handling

In chapter 2 we have seen that resilience requires us to distribute and compartmentalize our systems: distribution is the only way to avoid being knocked out by a single failure—be that hardware, software or human—and compartmentalization means to isolate the distributed units from each other such that the failure of one of them does not spread to the others. The conclusion was that in order to restore proper function after a failure, we need to delegate the responsibility of reacting to this event to a supervisor. It is worth remembering that validation errors are part of the normal operation protocol between modules while failures are those cases in which the normal protocol cannot be executed any longer and the supervision channel is needed—validation goes to the *user* of a service while failure is handled by the *owner* of the service.

The importance of ownership appeared already within the decomposition of our system according to *divide et regna* as we discussed the difference between a descendant module and a dependency. Descendants own a piece of the parent's functionality while foreign functions are incorporated only by reference. The resulting hierarchy is not only helpful for specification, test and scalability, it also gives us the supervision structure for our modules.

## 7.1 Ownership means Commitment

If a certain piece of the problem is owned by a module, then that means that this module needs to solve that part and provide the corresponding functionality because no other module will do it in its stead. Dependent modules will rely on this fact and will not operate correctly or to their full feature set when the module which is supposed to offer these function is not operational. In other words, ownership of a part of the problem implies a commitment to provide the solution because the rest of the application will depend on this.

Every function of an application is implemented by a module that owns it, and according to the hierarchical decomposition we have performed in the previous section

this module has a chain of ancestors reaching all the way up to the top level—that one corresponds to the high-level overall mission statement for the full system design as well as its top-most implementation module which is typically an application bundle or a deployment configuration manager for large distributed applications.

This ancestor chain is necessary because we know that failures will happen, and by failure we mean those incidents where a module cannot perform its function any longer (for example because the hardware it was running on stopped working). In the example of the Gmail "contacts" service, the low-latency search module could be deployed on several network nodes, and if one of them fails then the capacity of the system will be reduced unintentionally. The "search" service is the next owner in the ancestor chain, the *supervisor*, and since it is responsible for providing the functionality it will need to monitor the health of its descendant modules and initiate the start of new ones in case of failure. Only when that does not work—for whatever reason—does it signal this problem to its own supervisor.

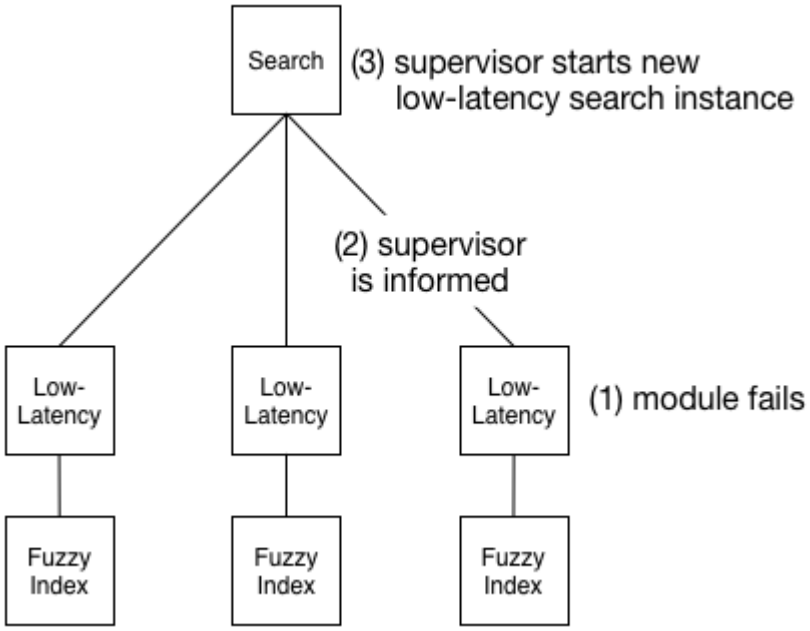


**Figure 7.1 a failure is detected and handled within the "contacts" module's part of the hierarchy, the "search" supervisor creates a new instance of the low-latency search in case one of them fails—for whatever reason.**

The rationale for this setup is that other parts of the application will depend on the "search" service and all the services it offers for public consumption, therefore it is this module that needs to ensure that its sub-modules function properly at all times. Responsibility is delegated downwards in the hierarchy to rest close to the point where each function is implemented, but the unpredictable nature of failure makes it necessary to delegate failure handling upwards when lower-level modules cannot cope with a given

situation. This works exactly as within an idealized (properly working) corporate structure, assuming that failure is treated as an expected fact of life and not swept under the carpet.

## 7.2 Ownership implies Lifecycle Control

The scheme of failure handling developed in the previous section implies that a module will need to create all sub-modules that it owns. The reason for this is that the ability to recreate them in case of a failure depends on more than just the spiritual ownership of the problem space, the supervisor must literally own the lifecycle of its sub-modules, it is its responsibility. Imagine some other module creating the low-latency search module and then handing it out to the “contacts” as a dependency, by reference. The “contacts” module would have to ask that other module to create a new low-latency search when it needs one, because that other module might not have realized that the old instance has failed. Replacing a failed instance includes clearing out all associated state like removing it from routing tables and in general clear all references to it so that the runtime can then reclaim the memory and other resources that the failed module occupied.

This has an interesting consequence in that the lifecycle of descendant modules is strictly bounded by its supervisor’s lifecycle—the supervisor creates it and without a supervisor it cannot continue to exist—while dependencies are not restricted in this fashion. In traditional dependency injection it is customary to create the dependencies first such that they are available when dependent modules are started up; similarly they are terminated only after all dependents have stopped using them. With dynamic composition as described in the section on location transparency this coupling becomes optional, dependencies can come and go at arbitrary points in time, changing the wiring between modules at runtime.

Considering the lifecycle relationship between modules therefore also helps in developing and validating the hierarchical decomposition of an application. Ownership implies a lifecycle bound while inclusion by reference allows independent lifecycles. The latter requires location transparency in order to enable references to dependencies to be acquired dynamically.

## 7.3 Resilience on All Levels

The way we deal with failure is inherently hierarchical; this is true in our society as well as in programming. The most common way to express failure in a computer program is by raising an exception, which is then propagated up the call stack by the runtime and delivered to the innermost enclosing exception handler that declares itself responsible. The same principle is expressed in pure functional programming by the return type of a called function in that it models either a successful result or an error condition (or a list thereof, especially for functions that validate input to the program). In both cases the responsibility of handling failure is delegated upwards, but in contrast to a the Reactive approach described above these techniques conflate the usage hierarchy with supervision—the user of a service gets to handle its failures as well.

Our principled approach to handling failure adds one more facet to the application's module hierarchy: every module is a unit of resilience. This is enabled by the encapsulation afforded by message passing and by the flexibility inherent in location transparency. A module can fail and be restored to proper function without its dependents needing to take action, the supervisor will handle this for everyone else's benefit.

This is true at all levels of the application hierarchy, although obviously the amount of work to be done during a restart depends on the fraction of the application which has failed. Therefore it is important to isolate failure as early as possible, keeping the units small and the cost of recovery low, which we will discuss in detail as the "Error Kernel Pattern" in chapter 12. But even in those cases where more drastic action is needed, the restart of for example the complete "contacts" service of our exemplary Gmail application will leave most of the "mail" functionality intact (searching, viewing and sorting mail does not depend critically on it, only convenience might suffer). This means that resilience can be achieved at all levels of granularity, and a reactive design will naturally lend itself well to this goal.

## 7.4 Summary

In this chapter we have utilized the hierarchical component structure of our application in order to determine a principled way of handling failure: by noting that the parent component is responsible for the functionality of its descendants it is logical to delegate the handling of such failures that cannot be dealt with locally to that same parent. This pattern allows the construction of software that is robust even in unforeseen cases and it is the cornerstone for implementing resilience.

In the next chapter we will investigate another aspect of building distributed components, which is that consistency comes at a price. Once again we will make use of the way we have split the problem into a hierarchy of independent components.

# 8 Delimited Consistency

One possible definition is that a *distributed system* is one whose parts can fail independently[66]. As motivated in chapter 2 and further detailed in the previous chapters we are describing a design that is distributed by its very nature: we want to model components that are isolated from each other and interact only via location transparent message passing in order to create a resilient supervisor hierarchy. This means that the resulting application layout will suffer the consequences of being distributed, foremost the loss of strong consistency as discussed at the end of chapter 2.

Footnote 66 A more humorous one by Leslie Lamport is: “A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.” (see https://research.microsoft.com/en-us/um/people/lamport/pubs/distributed-system.txt)

This can be illustrated on the example of the “mail” functionality of our exemplary Gmail application. Since the number of users is expected to be huge we will have to split the storage of all mail across many different computers located in multiple data centers distributed across the world. Assuming for now that a person’s folders can be split across multiple computers then that will mean that the act of moving an email from one folder to the other might imply that it moves between computers. Depending on how that is implemented it will either first be copied to the destination and then deleted at the origin, or it is placed in transient storage, deleted at the origin and then copied to the destination.

In either case the overall count of emails for that person should stay constant throughout the process, but if we count the emails by asking the computers involved in their storage then we might see the email “in flight” and count it either double or not at all. Ensuring that the count is consistent would mean to exclude the act of counting while the transfer is in progress. The cost of strong consistency is therefore that otherwise independent components of the distributed system need to coordinate their actions by way of additional communication, which means taking more time and using more

network bandwidth. This observation is not specific to the Gmail example but holds true in general for the problem of having multiple distributed parties agree on something—this is also called *distributed consensus*.

## 8.1 Encapsulated Modules to the Rescue

Fortunately these consequences are not as severe as they may seem at first. Pat Helland[67], a pioneer and long-time contributor to the research on strong consistency, argues that once a system's scale becomes large enough it cannot be strongly consistent any longer. The cost of coordinating a single global order of all changes that occur within it would be forbiddingly high, and adding more (distributed) resources will after that point diminish the system's capacity instead of adding to it. Instead, we will be constructing systems from small building blocks—*entities*[68]—that are internally consistent but interact in an eventually consistent fashion.

Footnote 67 see his paper on "Life beyond Distributed Transactions", http://www.ics.uci.edu/~cs223/papers/cidr07p15.pdf, published by CIDR 2007

Footnote 68 In the context of domain-driven design these would be called "aggregate roots"; the different uses of this word are owed to its intrinsic generality.

The dataset contained within such an entity can be treated in a fully consistent fashion, applying changes such that they occur—or at least appear to occur—in one specific order. This is possible because each entity lives within a *distinct scope of serializability*, which means that the entity itself is not distributed and that the datasets of different entities cannot overlap. The behavior of such a system is strongly consistent—*transactional*—only for those operations which do not span multiple entities.

He then goes on to postulate that we will develop platforms that manage the complexity of distributing and interacting with these independent entities, allowing the expression of business logic in a fashion that does not need to concern itself with the deployment details as long as it obeys the transaction bounds.

The entities Pat Helland talks about are very similar to the encapsulated modules we have developed in this chapter so far, the difference is mainly that he focuses on managing the data stored within a system while we concerned ourselves foremost with decomposing the functionality offered by a complex application. In the end both are the same: as viewed by a user the only thing that matters is that the obtained responses reflect the correct state of the service at the time of the request, where "state" is nothing else than the dataset which the service maintains internally—the downright embarrassingly simple case of a stateless service is discussed further in the next section. A system which supports reactive application design is therefore a natural substrate for fulfilling Pat Helland's prediction.

## 8.2 Grouping Data and Behavior According to Transaction Boundaries

The example problem of storing a person's email folders in a distributed fashion can be solved by applying the strategy outlined in the previous paragraphs. If we want to ensure that emails can be moved without leading to inconsistent counts, then each person's complete email dataset must be managed by one entity. In our exemplary application decomposition we would have a module for this purpose and we would instantiate it once for every person using the system. This does not mean that all mail is literally stored within that instance, it only means that all access to a person's email content is done via this dedicated instance.

In effect this will act like a locking mechanism that serializes access, with the obvious restriction that an individual person's email cannot be scaled out to multiple managers in order to support higher transaction rates. This is completely fine since a human is many orders of magnitude slower than a computer when it comes to processing email, so we will not run into performance problems by limiting scalability in this direction. What is more important is that this enables us to distribute the management of all users' mailboxes across any number of machines, because each instance is independent of all others. The consequence is that it is not possible to move emails between different persons' accounts while maintaining the overall email count, but that is not a supported feature anyway.

To formalize what we have just done, the trick is to slice the behavior and accompanying dataset in such a way that each slice offers the desired features in isolation and no transactions are necessary that span multiple slices. This technique is applied and discussed in great detail in the literature on Domain-Driven Design[69] (DDD).

---

Footnote 69 See for example "Domain-Driven Design" by Eric Evans, Addison-Wesley, 2003, or "Implementing Domain-Driven Design" by Vaughn Vernon, Addison-Wesley, 2013

---

## 8.3 Modeling Work-flows across Transactional Boundaries

The way in which we sliced the dataset accommodates a certain set of operations to be performed in a strongly consistent manner, but is precludes this quality of behavior for all the other conceivable operations. In most cases there will be operations that are desirable but which are not in the supported set, and slicing the data in another way is not an option because that would break more important use-cases. In this situation we must fall back to an eventually consistent way of performing that other operation, meaning that while we keep the atomicity of the transaction we punt on consistency and isolation.

To illustrate this we consider the case of moving an email from the mailbox of Alice to another person named Bob, possibly stored in a data center on another continent. While we cannot make this operation occur such that both the source and destination

mailboxes execute this operation “at the same time”, we can make sure that the email eventually will be present only in Bob’s mailbox. We facilitate this by creating an instance of a module that represents the transfer procedure, and this module will communicate with the mailbox instances for Alice and Bob to remove the email from the one and store it in the other. This so-called “Saga” pattern is shown in fig X and is discussed in detail in chapter 14.

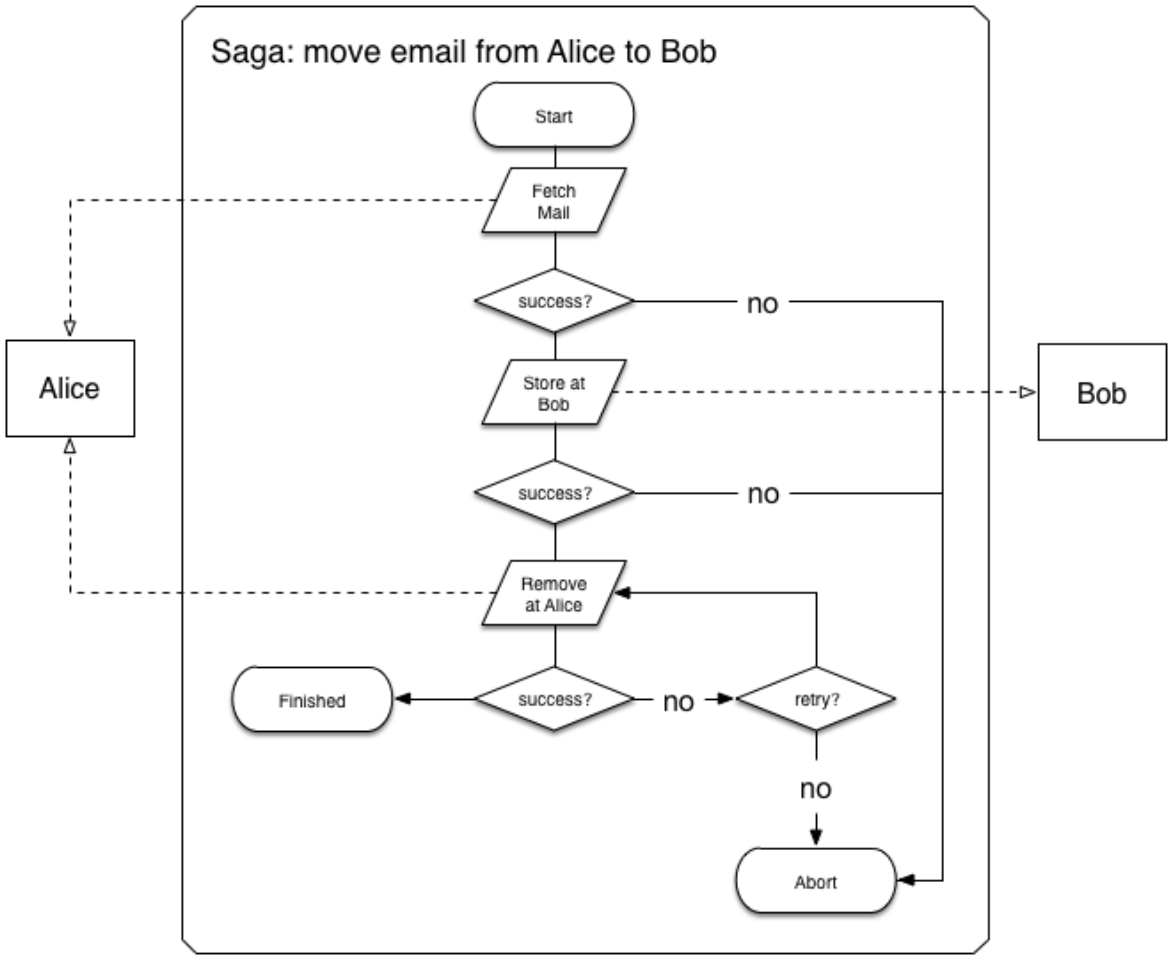


**Figure 8.1 a sketch of the Saga for moving an email from Alice’s account to Bob’s account, not including the cases for handling timeouts when communicating with the two accounts.**

Just as the mailbox modules will persist their state to survive failures, the Saga module can also be persistent. This ensures that even if the transfer is interrupted by a service outage it will eventually complete when the mailboxes for Alice and Bob are back online.

## 8.4 Unit of Failure = Unit of Consistency

Coming back to the initial definition of a distributed system at the beginning of this section, distributed entities are characterized by their ability to fail independently. Therefore the main concern of grouping data according to transactional boundaries was to ensure that everything that must be consistent is not distributed; a consistent unit must not fail partially, if one part of it fails then the whole unit must fail.

In the example of the transfer of an email between Alice's and Bob's mailbox the Saga which performs this task is one such unit. If one part of it fails then the whole transfer must fail, otherwise the email could be duplicated or vanish completely. This does not preclude the different subtasks from being performed by sub-modules of the Saga, but it requires that if one of the sub-modules fails then the whole Saga must fail as a whole.

Pat Helland's entities and our units of consistency therefore match up with the modules of the supervisor hierarchy which we have developed earlier in this chapter. This is another helpful property that can guide and validate the hierarchical decomposition of a system.

## 8.5 Segregation of Responsibilities

When breaking up a problem into smaller pieces we have postulated that we repeat this process iteratively until the parts we are left with are bite-sized and can efficiently be specified, implemented and tested. But what exactly is the right size? The criteria we have developed so far are

- a modules does one job, and does it well
- the scope of a module is bounded by the responsibility of its parent
- module boundaries define the possible granularity of horizontal scaling by replication
- modules encapsulate failure and their hierarchy defines supervision
- the lifecycle of a module is bounded by that of its parent
- module boundaries coincide with transaction boundaries

We have seen along the way that these criteria go hand in hand and are inter-related, meaning that abiding by one of them is likely to satisfy the others as well. Still we have a choice as to how big we want to make our modules. During the process of implementing and testing them—or with experience even during the design process—we might find that we did not choose wisely.

In case of a too fine-grained split we will notice that we need to use of messaging patterns like the Saga excessively often or that we have difficulty achieving the consistency guarantees that we require. The cure is relatively simple since the act of

combining the responsibilities of two modules just means that we compose their implementations, which is unlikely to lead to conflicts since the modules previously were completely independent and isolated from each other.

In the opposite case the we will suffer from complicated interplay of different concerns within a module, making supervision strategies difficult to determine or inhibiting necessary scalability. This defect is not as simple to repair as separating out different parts of the behavior means that we introduce new transaction boundaries between them. If the different parts become descendant modules then this might not have grave consequences because the parent module can still act as the entry point which serializes operations. But if the issue that prompted the split was insufficient scalability then this will not work since the implied synchronization cost (by way of going through a single funnel) was precisely the problem.

Segregating the responsibilities of such an object will necessarily require that some operations are relegated to eventually consistent behavior. One possibility that often applies is to separate the mutating operations (the *commands*) from the read operations (the *queries*). Greg Young coined the term *Command-Query Responsibility Segregation* [70] (CQRS) describing this split, which allows the write side of a dataset to be scaled and optimized independently from its read side. The write side will be the only place where modifications to the data are permitted, allowing the read side to act as a proxy that only passively caches the information that can be queried.

Footnote 70 For more material on this topic see http://www.cqrsinfo.com/.

Changes are propagated between modules by way of events—immutable facts that describe state change which have already occurred. In contrast the commands that are accepted at the write side merely express the intent that a change shall happen.

In the Gmail example we might have implemented the module that generates the overview of all folders and their unread email counts such that it accesses the stored folder data whenever it is asked for a summary to be displayed in the user's browser. The storage module will have to perform several functions:

- ingest new email as it arrives from the filtering module
- list all emails in a folder
- offer access to the raw data and meta-data for individual emails

The state of an email—for example whether it has been read or not—would naturally reside with the message itself, in the raw email object storage. One initial design might be to also store the folders to which it belongs together with each message, meaning that we obtain one fully consistent dataset for each user into which all emails are stored, and

this dataset is then queried in order to get the overview of read and unread message counts per folder. The corresponding query will have to traverse the meta-data of all stored emails and tally them according to folder name and status.

Doing it that way is rather costly, since the most frequently operation (checking for new email) will need to touch all of the meta-data, including old emails that have been read long ago. The additional downside is that ingesting new email will suffer from this mere observer function, since both kinds of activities will typically be executed by the storage module one after the other in order to avoid internal concurrency (and thereby non-determinism).

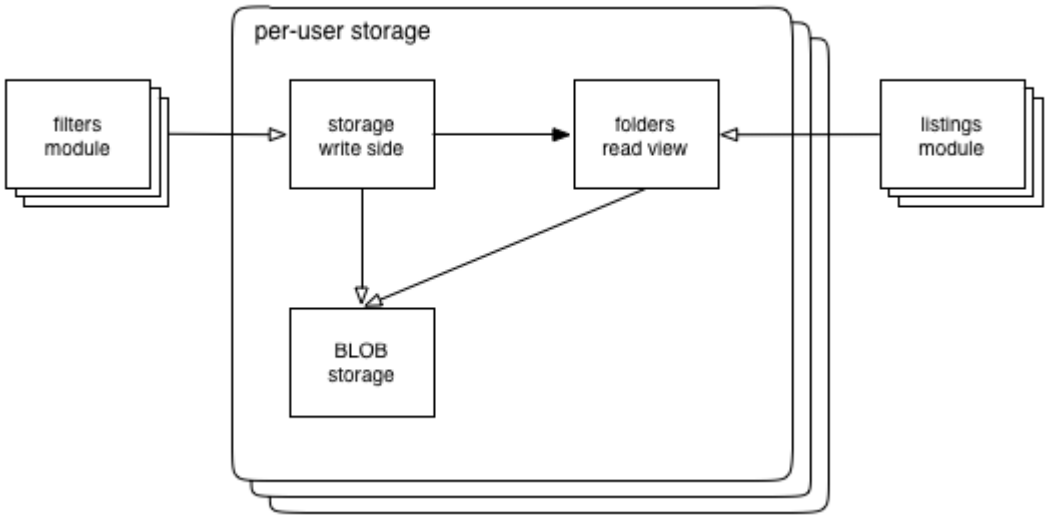


**Figure 8.2 per-user storage segregated into command and query responsibilities. New emails are written to the storage, informing the read view about the characteristics. Summary queries can be answered by the read view while raw email contents is retrieved directly from shared binary storage.**

This design can be improved by separating the responsibilities for updating and querying the email storage as shown in fig X. Changes to the storage contents, for example the arrival of new email but also adding and removing folder membership from messages or removing the “unread” flag, are performed by the write side, which persists these changes into the binary object storage. Additionally it informs the read view of the relevant meta-data changes so that this view can keep itself up to date about the read and unread email counts in each folder. This allows overview queries to be answered very efficiently without having to traverse the meta-data storage. In case of a failure the read view can always be regenerated by performing this traversal once, the view itself does not need to be persistent.

## 8.6 Persisting Isolated Scopes of Consistency

The topic of achieving persistence in systems designed for scalability is discussed in detail in chapter 16, but the application design described in this chapter has implications for the storage layer that deserve further elaboration. In a traditional database-centric application all data are held in a globally consistent fashion in a transactional data store. This is necessary since all parts of the application have access to the whole dataset and transactions can span various parts of it. Consequently a lot of effort is spent on defining and tuning the transactions at the application level as well as making the engine efficient at executing these complex tasks at the database level.

With encapsulated modules that each fully own their datasets these constraints are already solved during the design phase, what we would need is one database per module instance and each of these database would have to support modifications only from a single client. Almost all of the complexity of a traditional database would go unused with such a use-case since there will be no transactions to schedule or conflicts to resolve.

Coming back to CQRS we note that there is logically only one flow of data from the active instances to the storage engine: the application module sends information about its state changes in order to persist them. The only information it needs from the storage is the confirmation for successful execution of this task, upon which the module can acknowledge reception of the data to its clients and continue processing its tasks. This reduces the storage engine's requirements to just act as an append-only log of the changes—*events*—that the application modules generate. This scheme is called *event sourcing* because the persisted events are the source of truth from which application state can later be recovered.

An implementation with this focus will be much simpler and more efficient than using a transactional database since it does not need to support mutual exclusion of concurrent updates or even any form of altering persisted data at all. In addition, streaming consecutive writes is the operation for which all current storage technologies achieves the highest possible throughput.

Logging a stream of events for each module has the additional advantage that these streams contain useful information that can be consumed by other modules, for example updating a dedicated read view onto the data or providing monitoring and alerting functionality as discussed in chapter 16.

## 8.7 Summary

In this chapter we have discussed that strong consistency is not achievable across a distributed system, it is limited to smaller scopes and to units that fail as a whole. This has led us to add a new facet to our component structure, namely the recommendation to consider the business domain of the application in order to determine bounded contexts which are fully decoupled from each other—the terminology here is taken from Domain-Driven Design.

The driving force behind this search for a replacement of traditional transactionality and serializability stems from the non-determinism that is inherent to distributed systems. The next chapter places this finding into the larger context of the full range from logic programming and deterministic dataflow to full-on non-determinism experienced with threads and locks.

# 9 Non-Determinism by Need

*This section is the most abstract part of this chapter and it is not required for the initial understanding of the later sections, so you are welcome to skip ahead to "Message Flows" as long as you promise to come back here at a later time.*

In chapter 3 we introduced functional programming as one of the tools of the Reactive trade. The second part of this book has up to this point been concerned with splitting up a problem into encapsulated modules that interact only via asynchronous message passing, an act that is fundamentally impure: sending a message to an external object means that the state change of that object cannot be modeled within the sender, it necessarily is a side-effect[71], and this separation is not incidental, it is the entire reason why we compartmentalize our problem space.

Footnote 71 Tracking the sending of a message as an effect can of course be done, but this effect must always be executed in close proximity to the place where it originates since the modules we create in our hierarchical problem decomposition aim to be rather small. In pure functional programming the execution of effects is always deferred until the "end of the universe", but due to the aforementioned proximity the gain to be had by tracking this particular effect can be quite limited.

It seems therefore at first sight that the design we have chosen is fundamentally at odds with one of the core paradigms that we advertise. But this contradiction is not real as we will see now during a journey which could be titled "The Gradual Expulsion from Paradise".

## 9.1 Logic Programming and Declarative Data-Flow

The ideal of what we would want is that we specify the input data and the characteristics of the solution and the implementation of the programming language would do the rest for us. This would mean that we are free from the concerns of how the inputs are combined and processed, the computer would automatically figure out the right algorithm and perform it. This process would be fully deterministic as far as the formulated desired characteristics are concerned.

Research in this direction has led to the discipline of *logic programming* and the creation of programming languages like Prolog and Datalog in the 1980's. While not quite as advanced as the aforementioned ideal these languages allow the programmer to state the rules of a domain and ask the compiler to prove additional theorems which then correspond to the solution for a given problem. Logic programming so far has not had significant influence on mainstream software development, foremost due to its disadvantages with respect to runtime performance in comparison to imperative languages that are much farther removed from paradise.

The next step takes us to pure functional programming, expressing reasoning with functions and immutable values (where the former doubles as an example for the latter) in a fashion that is close to mathematics. Compared with the ideal we have traded decent runtime performance for the duty of figuring out the right algorithms ourselves: instead of generating the algorithm for us the compiler can at best verify that the algorithm we provided has the desired properties. The latter requires the use of a static type system that is powerful enough to encode the characteristics we want. In this camp we find a large number of languages to choose from, including Haskell, Coq, Agda and recently Idris. Many of the code samples in this book are written in the Scala language which can express pure functional programs but incorporates support for mutability and object-oriented as well.

Programming in a pure functional style means that the evaluation of every expression always yields the same result when given the same inputs, there are no side-effects. This enables the compiler to schedule evaluation on an as-needed basis instead of doing it strictly in the order given in the source code, including parallel execution. The necessary precondition for this is that all values are immutable, nothing can change after it has been created, with the very beneficial consequence that values can be freely shared among concurrently executing threads without any need for synchronization.

A close cousin of the functional paradigm is data-flow programming. The difference is that the former focuses on the functions and their composition while the latter concentrates on the movement of data through a network (more precisely a directed acyclic graph) of connected computations. Every node of this network is a single-assignment variable that is calculated once all inputs of its defining expressions have been determined. Therefore the result of injecting data into the processing network is always fully deterministic even though all computations within it conceptually run in parallel. Data-flow programming is for example part of the Oz language[72] but it can also be embedded in a language like Scala using composable Futures as shown in chapter 3.

Footnote 72 see http://mozart.github.io/mozart-v1/doc-1.4.0/tutorial/node8.html for more details on data-flow concurrency is realized

## 9.2 Functional Reactive Programming

A hybrid between the application of (pure) functions and the description of a processing network is *Functional Reactive Programming* (FRP), which focuses on the propagation and transformation of change, for example from measurements that arrive from a sensor to a GUI element on the human operator's screen. In its pure form FRP is rather close to data-flow programming in that it determines the changes to all input signals of a given transformation before evaluating it. This restricts implementations to run effectively single-threaded in order to avoid the problem of so-called *glitches*, which denotes the phenomenon that the output of an element fluctuates during the processing of an update throughout the network.

Recently the term FRP has been used also for implementations that are not glitch-free, like Rx.NET, RxJava and various JavaScript frameworks like React, Knockout, Backbone, etc. These concentrate on the efficient propagation of events and convenient wiring of the processing network, compromising on mathematical purity. As an example consider the following functions:

```
f(x) = x + 1
g(x) = x - 1
h(x) = f(x) - g(x)
```

In mathematical terms h(x) would always be exactly 2 because we can substitute the definitions of the other two functions into its body and witness that the only variable input cancels out. Written down in the aforementioned frameworks the result would most of the time be 2, but the values 1 and 3 would also be emitted from time to time (unless the user takes care to manually synchronize the two update streams for f and g).

This deviation from fully deterministic behavior is not random coincidence, and it is also not due to defects in these frameworks. It is a consequence of allowing the concurrent execution of effectful code, meaning code that manipulates the state of the universe surrounding the program. Both aspects—concurrency and effects—are essential to achieve good performance on today's hardware, making non-determinism a necessary evil. Another angle on this is presented in languages like Bloom[73] which employ the CALM[74] correspondence to identify those parts of a program which require explicit coordination, expressing the rest as so-called *disorderly programming*.

Footnote 73 See http://www.bloom-lang.net/features/ for an overview.

Footnote 74 Consistency and Logical Monotonicity, see "Relational transducers for declarative networking" by Ameloot et al, Journal of the ACM, Volume 60 Issue 2 (http://arxiv.org/pdf/1012.2858v1.pdf) for a formal treatment.

The attentive reader will have noticed that not all applications of the non-glitch-free

frameworks do indeed exhibit glitches. The only problematic operations are those which merge streams of updates that come from a common source and should therefore show a certain correlation. Processing networks which do not contain such operations will not suffer from non-determinism.

## 9.3 Shared-Nothing Concurrency

When concurrency as well as stateful behavior are required, there is no escape from non-determinism. This is obvious when considering the distributed execution of components that communicate via asynchronous message passing: the order in which messages from Alice and Bob reach Charlie is undefined unless Alice and Bob expend considerable effort in order to synchronize their communication[75]. The answer to Bob's question "Have you heard from Alice yet?" would thus vary unpredictably between different executions of this scenario.

Footnote 75 For example Alice could wait until Bob has heard back from Charlie—and told her so—before sending her message.

In the same way that distribution entails concurrency, the opposite is also true. Concurrency means that two threads of execution can make progress at the same time, independent of each other. In a non-ideal world this means that both threads can also fail independently, making them distributed by definition.

Therefore whenever a system comprises concurrent or distributed components there will be non-determinism in the interaction between these components. Non-determinism has a significant cost in terms of the ability to reason about the behavior of the program, and consequently we spend considerable effort on ensuring that all possible outcomes have been accounted for. In order to keep this overhead limited to what is required we want to bound the non-determinism we allow in our program to that which is caused by the distributed nature of its components, meaning that we only consider the unpredictability in the messaging sequence between encapsulated modules and forbid any direct coupling between them.

In this sense the term *shared-nothing concurrency* means that the internal mutable state of each module is safely stowed away inside of it and not shared directly with other modules. An example of what is forbidden would be to send a reference to a mutable object (for example a Java array) from one module to another while also keeping a reference. If both modules subsequently modify the object from within their transaction boundary their logic will be confused by the additional coupling that nothing to do with message passing.

The strategies to deal with non-determinism can be classified in two groups:

- We can reject those orderings of events which are problematic, introducing explicit synchronization to reduce the effect of non-determinism to a level where it does not change the program's characteristics any longer.

- Alternatively we can restrict the program to make use of operations that are commutative, which means that the order in which they are executed has no influence on the final result of the distributed computation[76].

Footnote 76 These data types are called Conflict-free Replicated Data Types (CRDT), see Shapiro et al, 2011 (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.231.4257&rep=rep1&type=pdf).

The former involves a runtime cost for coordination while the latter involves a development cost for restricting the exchanged dataset to be expressible efficiently in a conflict-free representation.

## 9.4 Shared-State Concurrency

The final step of your journey away from paradise brings us into the world of threads and locks and atomic CPU instructions, and it can be argued whether this corresponds to the necessary evil (i.e. the here and now) or whether this places us directly in hell. The background for this scenario is that our current computers are based on the Von Neumann architecture with the extension that multiple independent execution units share the same memory. The way data are transferred between CPU cores is therefore to read to and write from this shared memory instead of sending messages directly, with the consequence that all cores need to carefully coordinate their accesses.

Programming with threads and synchronization primitives maps quite directly to this architecture—we are glossing over the fact that threads are an illusion provided by the operating system to allow more concurrent executions than the number of available CPU cores—and it is the duty of the programmer to embed the right level of coordination in their program since the CPU would otherwise operate in its fastest possible and most reckless mode. The resulting code contains a tightly interwoven web of business logic and low-level synchronization, since memory accesses are so ubiquitous.

The problem with this code is that synchronization protocols do not compose well: if Alice knows how to conduct a conversation with Bob without getting confused, and Bob knows how to converse with Charlie, then that does not mean that the same way of talking with each other will allow a shared group conversation between all three of them. We also know from social experience that getting a larger group of people to agree on something is disproportionally more difficult than achieving agreement between just two persons.

## 9.5 So, What Should We Do?

Along the journey from paradise towards the netherworld we have gradually lost the ability to predict the behavior of our programs, towards the end it became nearly impossible to validate a design by reasoning about the way it is built up from simpler parts because the interplay between the building blocks entangles their internal behavior. This leads to the necessity of performing extensive testing and hopefully exhaustive verification scenarios for programs that are constructed directly upon threads and locks.

Threads and low-level synchronization primitives are an important tool for situations where performance requirements or the very nature of the problem (e.g. writing a low-level device driver) force us to exercise precise control over how the CPU instructions are carried out. This situation is one that most programmers will rarely find themselves in, we can in almost all cases rely on someone else to have solved that level of the problem for us; an example would be the implementation of an Actor framework that uses low-level features in order to provide users with a higher level of abstraction[77] [78].

Footnote 77 [a]as Doug Lea says, "I think about these things so that you don't have to."

Footnote 78 [b]yup; not sure about adding that, though

Retracing our steps we see that going from shared-state concurrency to shared-nothing concurrency we eliminate a whole class of problems from the application domain, we do not need to concern ourselves with the way CPUs synchronize their actions any longer since we write encapsulated components that only send immutable messages that can be shared without issues. Freed from this concern we can concentrate on the essence of distributed programming for solving our business problems, and this is where we want to be in case distribution is necessary.

The next step backwards removes the need for concurrency and distribution, enabling the resulting program to shed all non-determinism and greatly enhance the power of reasoning about the composition of a larger application from simple pieces. This is very desirable because it eliminates yet another class of defects from the application domain, we do not need to worry any more about manually having to ensure that things run in the right sequence. With FRP or data-flow we reason only about how data are transformed, not how the machine executes this transformation for us. Pure functional programming allows us to compose the application of calculations much in the same way that we would write them down mathematically—time does not play a role anymore[79].

Footnote 79 My favorite analogy is cooking: I know in principle how to cook all parts of a meal, and in practice I can also do it one part at a time (like concentrating on the meat until it is finished), but as soon as I try to do several things at once I start making mistakes. It would be so nice if time could be eliminated from this process, because that would allow me to do one thing after the other without the hassle of switching my focus all the time. This is quite like the difference between explicit concurrency and declarative programming.

Absent an efficient and proven implementation of logic programming this last step brought us to the place we would like to be in: a fully deterministic, reasonable programming model. The tools to program in this way are widely available, so we should make use of them wherever we can.

The dividing line between concurrent non-determinism and reasonable determinism is given by the need of distribution, a distributed system can never be fully deterministic because of the possibility of partial failure. The conclusion is therefore that employing a distributed solution must always be weighed against the cost associated with this, and apart from the distributed parts of a program you should always strive to stay as close to functional and declarative programming with immutable values as is feasible.

In the first sections of this chapter we have detailed the reasons for why and when distribution is necessary and useful, and the desire to stay non-distributed for as long as you can does not change this reasoning. The contribution of this section is to present a weight on the opposite side of resilience, scalability and responsibility segregation. We will have to place both on the scales and balance them well for each design that we develop. For further reading on this topic we recommend the literature detailing the design of the Oz language, in particular Peter van Roy's paper "Convergence in language design"[80].

Footnote 80 Peter van Roy, "Convergence in language design: a case of lightning striking four times in the same place", FLOPS'06, Proceedings of the 8th international conference on Functional and Logic Programming, pages 2–12 (https://mozart.github.io/publications/).

## 9.6 Summary

This chapter has taken us all the way from pure and deterministic programming approaches via shared-nothing concurrency to threads and locks. We have seen that all these tools have a place in our toolbelt and as we discussed we should be careful to stay as close to the initially mentioned paradigms as we can and only employ non-deterministic mechanisms where needed—be that for scalability or resilience. In the next chapter we close off the second part of this book by coming back to where we set out from in chapter 1: considering how messages flow through our applications.

# 10 Message Flow

Now that we have established a hierarchy of encapsulated modules that represent our application, we need to orchestrate these and realize the solution. The key point throughout the previous chapters has been that modules communicate only asynchronously by passing messages, they do not directly share mutable state. We have seen many advantages of this approach along the way, enabling scalability and resilience, especially in concert with location transparency, not to forget the fact that shared-state concurrency is hard to get right.

There is one further advantage that we will discuss in the following: basing a distributed design exclusively on messages allows us to model and visualize the business processes within our application as message flows and helps us to avoid limitations to scalability or resilience early in our planning process.

## 10.1 Push Data Forward

The fastest way for a message to travel from Alice via Bob to Charlie is if every station along the path sends it onwards as soon as it receives it. The only delays in this process are due to the transmission of the message between the stations and the processing of the message within each station.

As obvious as this statement is, it is instructive to consider the overhead added by other schemes. Alice could for example place the message in some shared storage and tell Bob about it; Bob would then retrieve the message from storage—possibly writing it back with some added information—and then tell Charlie about it, who now also looks at the shared storage. In addition to the two message sends we now have to perform three or four interactions with a shared storage facility, and we also remember that sharing mutable data between distributed entities is not the path to happiness.

Alice could also be concerned whether Bob currently has time for dealing with the message, so she asks him for permission to send it over. Bob will reply when ready, Alice will send the message and then the same procedure is repeated between Bob and

Charlie. Our two initial message sends are now accompanied by four more messages that convey readiness, first that of Alice to send more, then that of Bob to receive it.

Patterns like these are well established for purposes of persistence (e.g. durable message queues) or flow control (as we will discuss in depth in chapter 15) and they have their uses, but when it comes to designing the flow of messages through a reactive application it is important to keep the paths short and the messages flowing in one direction as much as possible: always towards the logical destination of the data. We will frequently need to communicate successful reception back to the sender, but that data stream can be kept lean by employing batching (cumulative acknowledgements).

The above examples were certainly simplistic, but the same principle applies more broadly. Coming back to our exemplary Gmail implementation that we have used throughout this chapter, the incoming emails that are sent to the system's users need to be transmitted from the SMTP module of the "mail" part of the application into the per-user storage. On their way there they will need to pass through the module which applies user-defined filters to sort each mail into the folders it belongs to.

As soon as the mails are in the per-user storage, they will be visible to the user in folder listings and so on, but in order to support a search function across the whole dataset owned by a user there will need to be an index kept up to date at all times. This index could periodically sync up with the current state of the mailbox storage and incorporate new emails, but that would be as inefficient as Bob periodically asking Alice whether there has been a new message since last time. Keeping the data flowing forward means in this case that the email will be sent in copy to the indexing service after having been classified by the filter module, which updates the index in real-time.

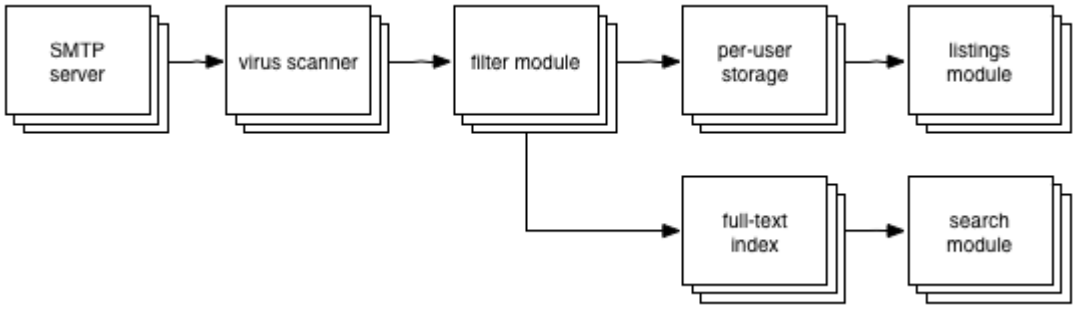


**Figure 10.1 data are flowing forward from their source (the SMTP server module) towards where the user needs them, feeding them to the indexing service in parallel to storing the raw data.**

In this fashion the number of messages that are exchanged is kept to a minimum and data are treated while they are "hot", which signifies both the relevance to the user as well as being in active memory in the computers involved. The alternative of polling every few minutes would ask the storage service for an overview of its data and thereby force it to keep this in memory—or read it back in after a resource shortage or outage.

## 10.2 Model the Processes of Your Domain

Programming with messages exchanged between autonomous modules also lends itself well to the use of *ubiquitous language* as practiced in domain-driven design. The customers of the software development process—which can be users or product owners—will be most comfortable in describing what they want in terms that they understand. This can be exploited for mutual benefit by turning the common language of the problem domain into modules and messages of the application architecture, giving concrete and rigorous definitions. The resulting model will be comprehensible for the customers as well as for the developers, and it will serve as a fix-point for communicating about the emerging product during the development process.

The reason behind this utility has been hinted at towards the end of chapter 2 already: anthropomorphic metaphors help us humans in visualizing and choreographing processes, it is an act that we enjoy, and this creates a fertile ground for finding ways of moving abstract business requirements into the realm of intuitive treatment. This was reason why we talked about Alice, Bob and Charlie instead of “node A, B and C”; in the latter case we would struggle trying to keep our reasoning technical while in the former case we can freely apply the wealth of social experience that we have accumulated. It is not surprising that we find good analogies for distributed computing within our society: we are the prototypical distributed system!

Our intuition is widely applicable in this process: when two facts need to be combined to perform a certain task then we know that there must be one person who knows both and combines them. This corresponds to the delimited consistency rule that we discussed above. Hierarchical treatment of failure is based on how our society works and message passing expresses exactly how we communicate. We should make use of these helpers wherever we can.

## 10.3 Identify Resilience Limitations

When laying out the message flows within our application according to the business processes that we want to model, we will see explicitly who needs to communicate with whom, or which module will need to exchange messages with what other module. Since we have also created the hierarchical decomposition of the overall problem and thus obtained the supervision hierarchy this will tell us which message flows are more or less likely to be interrupted by failure.

When sending to a module that is far down in the hierarchy and performing work that is intrinsically risky (like using an external resource) then we will have to foresee communication procedures for reestablishing the message flow after the supervisor has restarted the module. It can in some cases be better to send the messages via the supervisor from the start so that the clients—the senders of the messages—need not reacquire a reference to the freshly started target module so often. They will still need to

have recovery procedures in place to implement proper compartmentalization and isolation, but invoking them less frequently will further reduce the effect a failure has.

For this reason we will see some message flows that are directed from a module to its descendant, but in most cases the supervisor is only involved as a proxy while the real client is not part of the same supervision sub-tree. In general most message flows will be horizontal while supervision is performed on the vertical axis, coming back to the notion that usually the *user* and the *owner* of a service are not the same.

## 10.4 Estimate Rates and Deployment Scale

Focusing on message flows and sketching them out across the application layout allows us to make some educated guesses or apply input rates that we know from previous experience or measurement. As messages flow through the system and are copied, merged, split up and disseminated we can trace the associated rate information in order to obtain an impression which load the modules of the application will see.

When the first prototypes of the most critical modules are ready for testing we can then start evaluating their performance and use Little's formula to estimate the necessary deployment size as detailed in chapter 2. At this stage we will also be able to validate our assumption as to which modules need to be scaled out for performance reasons and where we can consolidate pieces that we split up erroneously.

The ability to perform these predictions and assessments stems from the fact that we have defined concrete units of work—messages—that we can count, buffer, spread out, and so on. We benefit from being explicit about message passing in our design. If we were to hide the distributed nature of our program behind synchronous RPC then this planning tool would be lost and we would be more concerned about trying to anticipate the size of the thread pools needed within our processes.

## 10.5 Plan for Flow Control

Closely related to the estimation process is that we need to foresee bulkheads between different parts of our application. When input message rates exceed the limits we planned for or when the dynamic scaling of our application is not fast enough to cope with a sudden spike in traffic, then there must be measures in place which contain the overflow and protect the other parts of the system.

With a clear picture of how increased message rates are propagated within the application we can for example determine at which points requests will simply be rejected (presumably close to the entrance of the application) and where we need to store messages on disk so that they are processed after the spike has passed or more capacity has been provisioned.

These mechanisms will need to be activated at runtime when their time comes, and this should be fully automatic because human response times are typically too long—especially on Sunday morning at 3am. We need to propagate the congestion

information upstream to enable the sender of a message stream to act upon it and refrain from overwhelming the recipient. Patterns for implementing this are discussed in chapter 15, of particular interest are Reactive Streams[81] as a generic mechanism for mediating back pressure in a distributed setting.

Footnote 81 see http://www.reactive-streams.org/

## 10.6 Summary for Part 2

In this and the previous chapters we have discussed the driving principles behind a reactive application design. The central piece is that we decompose the overall business problem in a hierarchical fashion according to *divide et regna* into fully encapsulated modules that communicate only by asynchronous, non-blocking and location-transparent message passing. The modularization process is guided and validated by the following rules:

- a modules does one job, and does it well
- the responsibility of a module is bounded by the responsibility of its parent
- module boundaries define the possible granularity of horizontal scaling by replication
- modules encapsulate failure and their hierarchy defines supervision
- the lifecycle of a module is bounded by that of its parent
- module boundaries coincide with transaction boundaries

We have then illuminated different paradigms ranging from logic programming to shared-state concurrency and concluded that we should prefer a functional and declarative style within these modules and consider the cost of distribution and concurrency when choosing the granularity of our modules.

Finally we noticed the advantages of explicitly modeling the message flows within our system for the purposes of keeping communication paths and latencies short, modeling our business processes using ubiquitous language, estimating rates and identifying resilience limitations and planning how to perform flow control.

# III Patterns

In the second part we discussed the building blocks upon which a reactive system can be built. It may be worthwhile to revisit that section while reading forward to ensure that the concepts it explained are clear in the implementation details we are about to discuss.

We have spent a fair amount of time so far discussing the what and the why of being reactive. Now it is time to focus on the how. In Part III, we present patterns of development that will help you implement reactive applications. We begin by discussing how to test that your application is reactive, so that you can build forward knowing that you are meeting the reactive contract, from the smallest components to an entire cross-data center deployment. Once we have covered that, we can delve into the specific patterns for building reactive systems across all dimensions of reactive concepts.

In this part you will learn:

- how to test reactive systems, with a specific emphasis on asynchronous testing
- how to layer internal and external fault tolerance into your application
- how to manage the resources utilized by your reactive application
- how to manage the flow of messages and data within and between your applications
- how to persist data in a reactive fashion
- how to use actors most effectively

# 11 Testing Reactive Applications

Now that the philosophy has been covered, we need to discuss how we can verify that the reactive applications we build are actually elastic, resilient and responsive. Testing is an early chapter because of the importance of proving our reactive capabilities. Just as Test Driven Design (TDD) allows the developer to ensure that they are writing logic that meets their requirements from the outset, we must focus on putting into place the infrastructure required to verify our elasticity, resilience and responsiveness. In this chapter, you will learn:

- how to approach the validation of asynchronous or non-deterministic systems
- how to measure responsiveness and elasticity
- how to test the resiliency measures that you put in place
- how to verify your failure handling

What we will not cover is how to test your business logic itself for there are countless good resources available on that topic. We will assume that you have picked a methodology and matching tools for verifying the local and synchronous parts of your application and focus instead on the aspect of distribution that is inherent to reactive systems.

## 11.1 How To Test

Testing applications is the foremost effort developers can undertake to ensure that code is written to meet all of its requirements without defects. Here a truly reactive application has several dimensions beyond merely fulfilling the specifications for the logic to be implemented, guided by the principles of Responsiveness, Elasticity and Resilience. In this book, where patterns are outlined to enable reactive applications, testing is integral to each pattern described so that you can verify that your application is reactive. In this chapter we lay the foundations for this by covering the common techniques and principles.

Before delving into patterns of testing, a vocabulary must be defined. For anyone who has worked for a mature development organization, testing is ingrained from the outset as a means to reduce risk. Consulting firms are also well-known for having stringent testing methodologies, in order to reduce the risk of a lawsuit from clients who have expectations about the level of quality for software being delivered. Every test plan is reviewed and approved by each level of the project leadership, from team leads through architects and project management with the ultimate responsibility residing with the partner or organizational stakeholder to ensure that accountability exists for any improper behavior that may occur. As a result, many levels of functional tests have been identified and codified into standards for successful delivery.

CALLOUT: In the following—in particular when touching on resilience—it is helpful to recall the distinction between errors and failures, as defined by the Glossary[1] of the Reactive Manifesto:

Footnote 1 http://www.reactivemanifesto.org/glossary#Failure

> A failure is an unexpected event within a service that prevents it from continuing to function normally. A failure will generally prevent responses to the current, and possibly all following, client requests. This is in contrast with an error, which is an expected and coded-for condition—for example an error discovered during input validation - that will be communicated to the client as part of the normal processing of the message. Failures are unexpected and will require intervention before the system can resume at the same level of operation. This does not mean that failures are always fatal, rather that some capacity of the system will be reduced following a failure. Errors are an expected part of normal operations, are dealt with immediately and the system will continue to operate at the same capacity following an error.

Examples of failures are hardware malfunction, processes terminating due to fatal resource exhaustion, program defects that result in corrupted internal state.

### *11.1.1 Unit Tests*

This is the best known of all of the kinds of tests, where an independent unit of source code, such as an object or class, is tested rigorously to ensure that every line and condition in the logic meets the specification outlined by the design or product owner. Depending on how the code is structured, this can be easy or hard - monolithic functions or methods inside of such a source unit can be difficult to test because of all of the varying conditions that can exist inside these units.

It is best to structure code into individual, atomic units of work that perform one action only. In doing so, writing unit tests for these units is quite simple - what are the expected inputs that should successfully result in a value against which assertions can be

made, and what are the expected inputs that should not succeed and result in an exception or error?

As unit tests focus on whether the right response is delivered for a given set of inputs, this level of testing will typically not involve testing for reactive properties.

### 11.1.2 Component Tests

These are also generally familiar to anyone who writes tests, in that we are now testing the Application Programming Interface (API) of a service. Inputs are passed to each public interface exposed by the API, and for several variations of correct input data it is verified that a valid response returned from the service.

Error conditions are tested by passing invalid data into each API and checking that the appropriate validation error is returned from the service. Validation errors should be an important design consideration for any public API, in order to convey an explicit error for any input that is deemed invalid for the service to handle appropriately.

Concurrency should also be explored at this level, where a service that should be able to handle multiple requests simultaneously returns the correct value for each client. This can be difficult to test for systems that are synchronous in nature, as concurrency in this case can involve locking schemes and a difficult setup on the part of the person writing the test to prove that multiple requests are being handled at the same time.

At this level we also start testing the responsiveness of a service to see whether it can reliably keep its SLA under nominal conditions, and for a component that acts as a supervisor for another we will also encounter aspects of resilience: does the supervisor react correctly to unexpected failures in its subordinates?

### 11.1.3 String Tests

Now we diverge from the ordinary practices of testing, where we need to verify that requests into one service or microservice that depends on other such services or microservices can return the appropriate values. It is important not to get bogged down in the low-level details of functionality that has already been tested at the unit and component test levels.

At this level we also start to consider failure scenarios in addition to the nominal and error cases: how should a service react to the inability of its dependencies to perform their function? It is also important to verify that SLAs are kept when dependent services take longer to respond than usual, both within and without their respective SLAs.

### 11.1.4 Integration Tests

Typically, the systems that we build do not exist in a vacuum, and prior to this level of testing, dependencies on external components are stubbed out or mocked so that we do not require access to these systems to prove that everything else in our system meets our requirements. However, when we reach the Integration testing level, we do want to ensure that such interactions are proven to work and handle nominal as well as erroneous input as expected.

At this level we also test for resilience by injecting failures, for example through shutting off services to see how their communication partners and supervisors react. We also need to verify that SLAs are kept under nominal as well as failure scenarios and under varying degrees of external load.

### 11.1.5 User Acceptance Tests

This is a final level of testing that is not always explicitly executed, most notable exceptions include situations where the consequences of failing to meet the requirements are severe (e.g. space missions, high-volume financial processing, military applications). The purpose is to prove that the overall implementation meets the project goals set by the client paying for the system to be built. But they can be applicable for organizations who treat the product owner as a client and the delivery team as the consulting firm. User acceptance testing is the level where the check-writer defines proofs that the system meets their needs, in isolation from those tests implemented by the consulting firm themselves. This provides independent verification that the application, with all of its various components and services, fulfills the ultimate goal of the project.

This level of testing may sound unnecessary for projects where an external contractor or firm has been hired to build the implementation, but we argue otherwise. One of the great benefits of test tooling such as Behavior Driven Development (BDD)[2] is to provide a Domain Specific Language (DSL)[3] for testing that even non-technical team members can read or even implement. Using tools such as well-known implementations and variants of Cucumber[4] (like Cuke4Duke[5], Specs2[6] and ScalaTest[7]) provides business process leaders on teams with the capability to write and verify tests.

Footnote 2 http://en.wikipedia.org/wiki/Behavior-driven_development

Footnote 3 http://en.wikipedia.org/wiki/Domain-specific_language

Footnote 4 http://cukes.info/

Footnote 5 https://github.com/cucumber/cuke4duke

Footnote 6 http://etorreborre.github.io/specs2/

Footnote 7 http://www.scalatest.org/

### 11.1.6 Black Box Versus White Box Tests

When testing a component there is a decision to be made on whether the test shall have access to the internal details of it or not. Such details include being able to send commands that are not part of the public interface or to query the internal state that is normally encapsulated and hidden. Testing without access to these is called *black box testing* because the component is viewed as a box that hides its inner workings in darkness. The opposite is termed *white box testing* because all details are laid bare like in a laboratory clean room where all internals can be inspected.

A reactive system is defined by its responses to external stimulus, which means that testing for the reactive properties of your applications or components will focus on the former kind, even if you prefer to use white box testing within the unit tests for the business logic itself.

As an example you might have a minute specification that is very precise about how the incoming data are to be processed, and the algorithm is implemented such that intermediate results can be inspected along the way. The unit tests for this part of the application will be tightly coupled to the implementation itself and for any change made to the internals there is a good chance that some test cases are invalidated.

This piece of code will form the heart of your application, but it is not the only part: data need to be ingested for processing, the algorithm must be executed and results need to be emitted, and all these aspects require communication and are governed by the reactive principles. You will hence write other tests that verify that the core algorithm is executed when appropriate and with the right inputs and that the output arrives at the desired place after the allotted time in order to keep the SLA[8] for the service you are implementing. All these aspects do not depend on the internal details of the core algorithm, they operate without regard to its inner workings. This has the added benefit that the higher-level tests for responsiveness, elasticity and resilience will have a higher probability of staying relevant and correct while the core code is being refactored, bugs are fixed or new features are added.

---

Footnote 8 Service Level Agreement

---

## 11.2 Test Environment

A very important consideration for writing tests is that they must be executed on hardware that is at least somewhat representative of that on which it will ultimately be deployed, particularly for systems where latency and throughput must be validated (to be discussed later in the “Testing Responsiveness” section) . This may provide some insight into how well a component or algorithm may perform, particularly if the task is CPU-intensive, relative to another implementation tested on the same platform.

Many popular benchmarks in the development community are run on laptops,

machines with limited resources with respect to number of cores, size of caches and memory, disk subsystems that do not perform data replication and operating systems that do not match intended production deployments. A laptop is typically constructed with different design goals than a server class machine, leading to different performance characteristics where some activities may be performed faster and others slower than on the final hardware. While the laptop may have capabilities that actually exceed a specific server-class machine (for example, a solid-state drive as opposed to a hard disk for storage) that makes it perform better in certain situations, it will likely not represent the performance to be expected when the application reaches production. Basing decisions on the results of tests executed in such an environment may lead to poor decisions being made about how to improve the performance of an application.

It is an expensive proposition to ask all companies, particularly those with limited financial resources such as startups, to consider mirroring their production environment for testing purposes. However, the cost of not doing so can be enormous if an organization makes a poor choice based on meaningless findings derived from an development environment.

Note that deployment in the cloud can make testing more difficult as well. Hypervisors do not necessarily report accurately about the resources they make available to us in a multi-tenancy environment, particularly with respect to the number of cores available to our applications at any given time. This can make for highly dynamic and unpredictable performance in production. Imagine trying to size thread pools in a very specific way for smaller instances in the cloud where you are not expecting access to more than 4 virtual CPUs, but there is no guarantee you will receive that at any given moment. If you must verify specific performance via throughput and/or latency, dedicated hardware is a considerably better option.

## 11.3 Testing Asynchronously

The most prominent difficulty that arises when testing reactive systems is that the pervasive use of asynchronous message-passing requires a different way of formulating test cases. Consider testing a translation function that can turn Swedish text into English:

```
val input = "Hur mår du?"
val output = "How are you?"
translate(input) should be(output)
```

This example uses ScalaTest syntax, in the first two lines we define the expected input and output strings and the third line then invokes the translation function with the input and asserts that this should result in the expected output. The underlying assumption is that the translate() function computes its value synchronously and that it is done when the function call returns.

A translation service that can be replicated and scaled out will not have the possibility of directly returning the value, it will have to be able to asynchronously send the input string to the processing resources. This could be modeled by returning a Future[9] for the result string that will eventually hold the desired value:

Footnote 9 Recall that a Future is a handle to a value that may be delivered asynchronously at a later time. The code that supplies the value will fulfill the corresponding Promise with it, enabling those who hold the Future to react to the value using callbacks or transformations, flip back to chapter 2 to refresh the details if necessary.

```
val input = "Hur mår du?"
val output = "How are you?"
val future = translate(input)
// what now?
```

The only thing that we can assert at this point is that the function does indeed return a future, but very likely there will not yet be a value available within it so we cannot continue with the test procedure.

Another presentation of the translation service might make use of Actor messaging, which means that the request is sent as a one-way message and the reply is expected to be sent as another one-way message at a later point in time. In order to receive this reply, there needs to be a suitable recipient:

```
val input = "Hur mår du?"
val output = "How are you?"
val probe = TestProbe() // an Akka utility
translationService ! Translate(input, probe.ref)
// when can we continue?
```

The TestProbe is an object that contains a message queue to which messages can be sent via the corresponding ActorRef. We are using that one as the return address in the message to the translation service Actor. Eventually the service will reply and the message with the expected output string should arrive within the probe, but again we cannot proceed with the test procedure at this point since we do not know when exactly that will be the case.

### 11.3.1 Providing Blocking Message Receivers

*Callout: The methods used in implementing the solutions below are typically not recommended for regular use because of their thread-blocking nature, but bear with us: even for testing we will present nicely non-blocking solutions later-on. Usage of classical test frameworks can require us to fall back to what is discussed here and it is educational to consider the progression presented in this section.*

One solution to the dilemma is to suspend the test procedure until the translation service has performed its work and then inspect the received value. In case of the Future

we can poll its status in a loop:

```
while (!future.isCompleted) Thread.sleep(50)
```

This will check every 50 milliseconds whether the Future has received its value or was completed with an error, not letting the test continue before that is the case. The syntax used is that of scala.concurrent.Future, in other implementations the name of the status query method could be isDone() (java.util.concurrent.Future), isPending() (JavaScript Q) or inspect() (JavaScript when) to name a few examples. In a real test procedure the number of loop iterations would have to be bounded:

```
var i = 20
while (!future.isCompleted && i > 0) {
  i -= 1
  Thread.sleep(50)
}
if (i == 0) fail("translation was not received in time")
```

This will wait only for up to roughly one second and fail the test if the Future is not completed within that time window. Otherwise message loss or a programming error could lead to the Future never receiving a value and then the test procedure would just hang and never yield a result.

Most Future implementations include methods that support awaiting a result synchronously, a selection is shown in Table 11.1.

**Table 11.1 Methods for synchronously awaiting a Future result**

| Language | Synchronous Implementation |
|---|---|
| Java | `future.get(1, TimeUnit.SECONDS);` |
| Scala | `Await.result(future, 1.second)` |
| C++ | `std::chrono::milliseconds span(1000);`<br>`future.wait_for(span);` |

These can be used in tests to retain the same test procedure as for the verification of the synchronous translation service:

```
val input = "Hur mår du?"
val output = "How are you?"
val result = Await.result(translate(input), 1.second)
result should be(output)
```

With this formulation we can take an existing test suite for our translation service and mechanically replace all invocations that used to return a strict value into synchronously awaiting the value using the returned Future. Since the test procedure is typically executed on its own dedicated thread this should not interfere with the implementation of the service itself on its own.

It must be stressed that this technique is very likely to fail if applied outside of testing and within production code. The reason is that the caller of the translation service will then no longer be an isolated external test procedure, it will most likely be another service that may use the same asynchronous execution resources. If enough calls are made concurrently in this thread-blocking fashion then all threads of the underlying pool will be parked, idly waiting for the responses, and the desired computation will not get executed since no thread will be available to pick it up. Timeouts or deadlock will ensue.

Coming back to the test procedures we still have one open question: how does this work for the case of one-way messaging as in the Actor example? We used a TestProbe as the return address for the reply. Such a probe is equivalent to an Actor without processing capabilities of its own which provides utilities for synchronously awaiting messages. The test procedure would in this case look like the following:

```
val input = "Hur mår du?"
val output = "How are you?"
val probe = TestProbe()
translationService ! Translate(input, probe.ref)
probe.expectMsg(1.second, output)
```

The expectMsg() method will wait for up to one second for a new message to arrive and if that happens it will compare it to the expected object—the output string in this case. If nothing or the wrong message is received then the test procedure will fail with an assertion error.

### 11.3.2 The Crux with Choosing Timeouts

Most synchronous test procedures verify that a certain sequence of actions results in a given sequence of results: we set up the translation service, we invoke it and compare the returned value to the expected one. This means that the aspect of time does not play a role in these tests at all, the test result will not depend on whether running it is a matter of milliseconds or takes a few hours. The basic assumption is that all processing occurs in the context of the test procedure, literally beneath its frame of control. Therefore it is enough to react to returned values or thrown exceptions, there will always be a result—infinite loops will be noticed eventually by the human observer.

In an asynchronous system this assumption does not hold any longer: it is possible that the execution of the module under test occurs far removed from the test procedure and replies might not only arrive late, they can also be lost. The latter can be due to

programming errors (not sending a reply message, not fulfilling a Promise in some edge case, etc.) or it can be due to failures like message loss on the network or resource exhaustion—if an asynchronous task cannot be enqueued to be run then its result will never be computed.

For this reason it is unwise to wait indefinitely for replies during test procedures, since we do not want the whole test run to grind to a halt half-way through just because of one lost message. We need to place an upper bound on waiting times, fail tests which violate it and move on.

This upper bound should be long enough to allow natural fluctuations in execution times without leading to sporadic test failures; such flakiness would waste resources during development in order to investigate each test failure whether it was legitimate or bad luck. Typical sources of bad luck include garbage collection pauses, network hiccups, temporary system overload, and all of these can cause a message send that normally just takes microseconds to be delayed by up to several seconds.

On the other hand the upper bound needs to be as low as possible since it defines the time it takes to give up and move on and we do not want to wait for a verification run to take several hours when one hour would suffice.

### SCALING TIMEOUTS FOR DIFFERENT TEST ENVIRONMENTS

Choosing the right timeouts is therefore a compromise between worst-case test execution time and false positive error probability. On current notebook computers as the one I am using right now it is realistic to expect asynchronous scheduling to occur on the scale of tens of milliseconds; normally it happens much faster but if we are for example executing a large test suite with thousands of tests on the JVM we need to take into account that the garbage collector will occasionally run for a few milliseconds and we do not want that to lead to test failures since it is expected behavior for a development system.

If we develop a test suite this way and then let it be run on a continuous integration server in the cloud we will discover that it fails miserably. The server will likely share the underlying hardware with other servers through virtualization, and it might also perform several test runs simultaneously. These and other effects of not having exclusive access to hardware resources lead to greater variations in the execution timing and thereby force us to relax the expectations as to when processing should occur and replies should be received.

To this end many asynchronous testing tools as well as the test suites themselves contain provisions to adapt the given timeouts to different runtime environments. In it simplest form this just means scaling by a constant factor or adding a constant amount to account for the expected variance.

*Callout: This is realized within the ScalaTest framework by mixing in the trait* ScaledTimeSpans *and overriding the method* spanScaleFactor(). *Another example is the*

*Akka test suite which allows the external configuration of a scaling factor that is applied to durations used in* TestProbe.expectMsg() *and friends (the configuration key is* akka.test.timefactor*).*

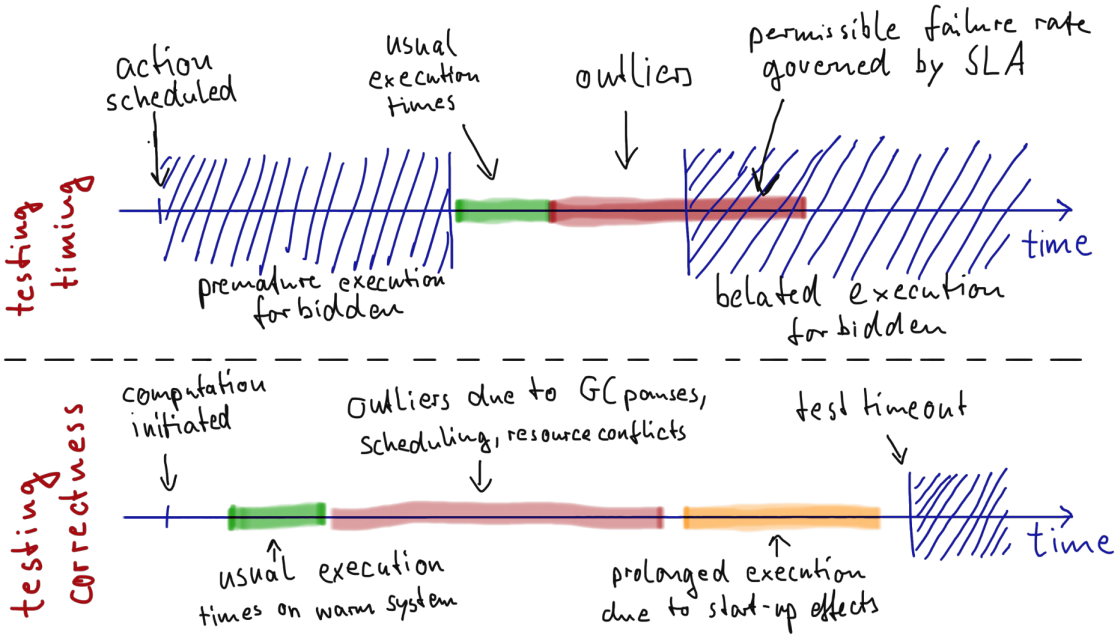


**Figure 11.1 Testing a system for correctness and testing it for its timing properties are significantly different activities.**

## TESTING SERVICE TIMINGS

Another issue can arise with testing asynchronous services: due to the inherent freedom of when to reply to a request we can imagine services that shall only reply after a certain time has passed or that shall trigger the periodic execution of some action. All such use-cases can be modeled as external services that will arrange for messages to be sent at the right times so that other services can depend on them for their scheduling needs.

The difference between testing the timing behavior of a service versus using timeouts for verifying its correctness is illustrated in Figure 11.1. If we just want to assert that the right answer is received, we will choose a timeout that bounds the maximal waiting time such that normally the test succeeds even if the execution is delayed much longer than would be expected during production use. Testing a service for its timing shifts the aspect of time from a largely ignored bystander role into the center of the focus, we now need to put more stringent limits on what to accept as valid behavior, and we may need to establish lower bounds as well.

How do we implement a test suite for a scheduler? As an example we formulate a test case for a scheduler service that is implemented as an Actor, using again a TestProbe as the communication partner that is controlled by the test procedure:

```
val probe = TestProbe()

val start = Timestamp.now
scheduler ! Schedule(probe.ref, "tick", 1.second)
probe.expectMsg(2.seconds, "tick") // check that it arrives at all
val stop = Timestamp.now

val duration = stop - start
assert(duration > 950.millis, "tick came in early")
assert(duration < 1050.millis, "tick came in late")
```

Here verification proceeds in two steps. Firstly we verify that the scheduled message does indeed arrive, using a relaxed time constraint with similar reasoning as for the timing-agnostic tests discussed in the previous section. Secondly we also take note of the time that elapsed between sending the request and receiving the scheduled message and assert that this time interval matches the requested schedule.

The second part is subject to all the timing variations due to external influences that we considered earlier, which poses a problem. We cannot evade the issues by relaxing the verification constraints this time because that would defeat the purpose of the test. This leaves only one way forward: we need to run these timing-sensitive tests in an environment that does not suffer from additional variances. Instead of including it with all the other test suites that we run on the continuous integration servers we might choose to only execute them on those reliable and fast developer machines which work much better in this regard.

But this would also be problematic in that a scheduler which passes these local tests under ideal circumstances could fail to meet its requirements when deployed in production. Therefore such services need to be tested in an environment which closely matches the intended production environment, both in terms of the hardware platforms used and the kind of processes running simultaneously and their resource configurations. It will make a difference if the timing-sensitive service commands independent resources or shares a thread pool with other computation.

### TESTING SERVICE LEVEL AGREEMENTS

In chapter one we discussed the importance of establishing reliable upper bounds for service response times in order to conclude whether the service is currently working or not. In other words each service needs to abide by its Service Level Agreement (SLA) in addition to performing the correct function. The test procedures we have seen so far only concentrated on verifying that a given sequence of actions produces the right sequence of results, where in certain cases the result timing might be constrained as well. For verifying the SLA it is necessary to test aspects like the 95th percentile of the request latency, for example asserting that it must be below 1 millisecond. These tests are inherently statistical in nature, necessitating additions to our set of testing tools.

Formulating test cases concerned with latency percentiles for a given request type means that we will need to perform such requests repeatedly and keep track of the time elapsing between each matching request–response pair. The simplest way is to do this is to sequentially perform one request after the other.

```
val probe = TestProbe()
val echo = echoService("keepSLA") // obtain a service ActorRef
val N = 200
val timings = for (i <- 1 to N) yield {
  val string = s"test$i" // generate strings test1, test2, …
  val start = Timestamp.now
  echo ! Request(string, probe.ref)
  // include a hint which step failed in case of timeout
  probe.expectMsg(100.millis, s"test run $i", Response(string))
  val stop = Timestamp.now
  stop - start // return the elapsed time for this run
}
// discard top 5%
val sorted = timings.sorted
val ninetyfifthPercentile = sorted.dropRight(N * 5 / 100).last
ninetyfifthPercentile should be < 1.millisecond
```

This test procedure takes note of the response latency for each request in a normal collection which is then sorted in order to extract the 95th percentile (by dropping the highest 5% and the looking at the largest element). This shows that no histogramming package or statistics software is necessary to perform this kind of test, so there is no excuse to skimp on such tests. In order to learn more about the performance characteristics and dynamic behavior of the software we write it is recommended, though, to visualize the distribution of request latencies; this can be done for regular test runs or in dedicated experiments and statistics tools will help in this regard.

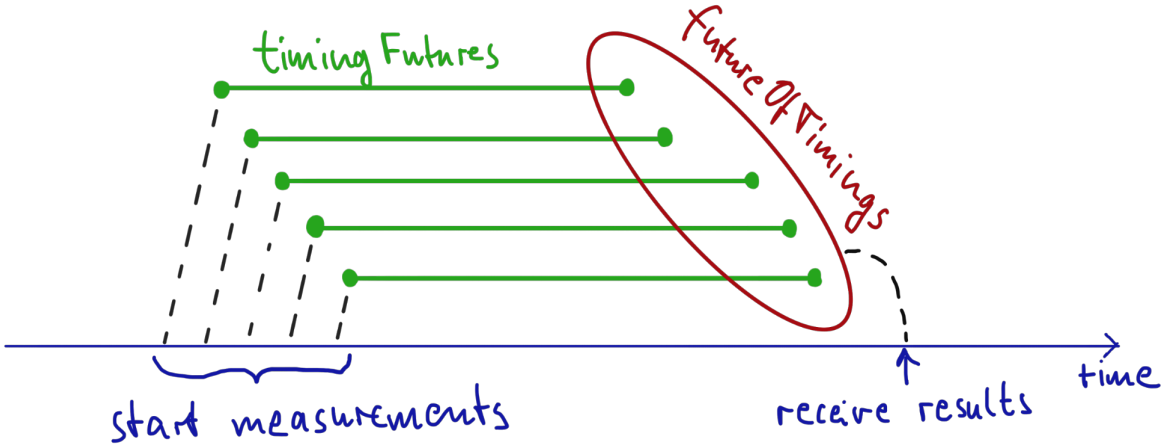


**Figure 11.2 The test procedure initiates multiple calls to the service under test which may be executed in parallel, aggregates the timings and verifies that the SLA is met.**

In the presented test case requests are fired one by one, the service will not experience

any load during this procedure. The obtained latency values will therefore reflect the performance under ideal conditions, it is likely that under nominal production conditions the timings will be worse. In order to simulate a higher incoming request rate—corresponding to multiple simultaneous uses of the same service instance—we need to parallelize the test procedure as shown in Figure 11.2. The easiest way to do this is to utilize Futures:

```
val echo = echoService("keepSLAfuture")
val N = 10000
val timingFutures = for (i <- 1 to N) yield {
  val string = s"test$i"
  val start = Timestamp.now
  // using the ask-pattern, see below
  (echo ? (Request(string, _))) collect {
    case Response(`string`) => Timestamp.now - start
  }
}
val futureOfTimings = Future.sequence(timingFutures)
val timings = Await.result(futureOfTimings, 5.seconds)
// discard top 5%
val sorted = timings.sorted
val ninetyfifthPercentile = sorted.dropRight(N * 5 / 100).last
ninetyfifthPercentile should be < 100.milliseconds
```

This time we use the ? operator to turn the one-way Actor message send into a request–reply operation: this method internally creates an ActorRef that is coupled to a Promise and uses the passed-in function to construct the message to be sent. Scala's function literal syntax makes this very convenient using the underscore shorthand—you can mark out the "hole" into which the ActorRef will be placed. The first message sent to this ActorRef will fulfill the Promise and the corresponding Future is returned from the ? method (pronounced "ask").

We then transform this future using the collect combinator: in case it is the expected response we replace that with the elapsed time; here it is essential to remember that Future combinators execute when the Future is completed, in the future, and hence taking a timestamp within the collect transformation serves as the second look to the watch, while the result of the first look was obtained from the test procedure's context and stored in the start timestamp that we then later reference from the Future transformation.

The for-comprehension returns a sequence of all Futures which we can turn into a single Future holding a sequence of time measurements by using the Future.sequence() operation. Synchronously awaiting the value for this combined future lets us then continue in the same fashion as for the sequentially requesting test procedure.

If you execute this parallel test you will notice that the timings for the service are markedly changed for the worse. This is because we very rapidly fire a burst of requests

which will pile up in the EchoService's request queue and then processed one after the other. Therefore on my machine I had to increase the threshold for the 95th percentile to 100 milliseconds, otherwise I experienced spurious test failures.

Just as the fully sequential version exercised an unrealistic scenario the fully parallel one tests a rather special case as well. A more realistic test would be to limit the number of outstanding requests to a given number at all times, keeping for example 500 in flight. Formulating this with Futures will be very tedious and complex[10], in this case it is preferable to call upon another message-passing component for help. In this example we will use an Actor to control the test sequence:

Footnote 10 Assuming that the responses can arrive in a different order than the one we sent the corresponding requests in; this assumption is necessary to make for a service that can be scaled by replication.

```
val echo = echoService("keepSLAparallel")
val probe = TestProbe()
val N = 10000
val maxParallelism = 500
val controller = system.actorOf(Props[ParallelSLATester],
                                "keepSLAparallelController")
controller ! TestSLA(echo, N, maxParallelism, probe.ref)
val result = probe.expectMsgType[SLAResponse]
// discard top 5%
val sorted = result.timings.sorted
val ninetyfifthPercentile = sorted.dropRight(N * 5 / 100).last
ninetyfifthPercentile should be < 2.milliseconds
```

The code for the Actor itself can be studied in the accompanying source code archives, the idea is to send the first maxParallelism requests when starting the test, then sending one more for each received response until all requests have been sent. For each request that is sent a timestamp is stored together with the unique request string and when the corresponding response is received the current time is used to calculate this request's response latency. When all responses have been received a list of all latencies is sent back to the test procedure within an SLAResponse message. From there on the calculation of the 95th percentile proceeds as usual.

**SIDEBAR** Looking at the code in the source archive you will notice that the above is slightly simplified: instead of directing the responses to the ParallelSLATester a dedicated Actor is used which timestamps the responses before sending them on to the ParallelSLATester. The reason is that otherwise the timings might be distorted because the ParallelSLATester might still be busy sending requests when a response arrives, leading to an artificially prolonged time measurement. Another interesting aspect is the thread pool configuration, you are welcome to play with the parallelism-max setting to find out when the results are stable across multiple test runs and when they become optimal; for a discussion see the comments in the source code archives.

### *11.3.3 Asserting the Absence of a Message*

All verification we have done so far concerned messages that were expected to arrive, but it is equally important to verify that certain messages are not sent. When components interact with protocols that are not purely request–response pairs this arises frequently:

- after cancelling a repeating scheduled task
- after unsubscribing from a Publish–Subscribe topic
- after having received a data set that was transferred via multiple messages

Depending on whether the messaging infrastructure maintains the ordering for messages traveling from sender to recipient we can either expect the incoming message stream to cease immediately after having confirmation that the other side will stop or we need to allow for some additional time during which stragglers may still arrive. In a test procedure like the ones above the absence of a message can only be asserted by letting a certain amount of time elapse and verifying that indeed nothing is received during this time.

```
val probe = TestProbe()
scheduler ! ScheduleRepeatedly(probe.ref, 1.second, "tick")
val token = expectMsgType[SchedulerToken]
probe.expectMsg(1500.millis, "tick")
scheduler ! CancelSchedule(token, probe.ref)
probe.expectMsg(100.millis, ScheduleCanceled)
// now we don’t expect any more ticks
probe.expectNoMsg(2.seconds)
```

Looking at the expected message timings and summing them up this procedure should take a bit more than three seconds: one for the first tick to arrive, some milliseconds for the communication with the scheduler service and two more seconds during which we do nothing at all. Verifications like this one will increase the time

needed to run the whole test suite, usually even more than most tests that spend their time more actively. It is therefore desirable to reduce occurrences of this pattern as much as possible.

One way to achieve this is to rely upon message ordering guarantees where available. Imagine a service implementing data ingestion and parsing: we send it a request that points it to an accessible location—a file or a web resource—and we will get back a series of data records followed by an end-of-file marker. Each instance of this service would process requests in a purely sequential fashion, finishing one response series before picking up the next work item. This will make the service easier to write and scaling it out will be trivial by running multiple instances in parallel; the only externally visible effect is that requests need to contain a correlation ID since multiple series can be in flight at the same time. A test procedure demonstrates the interface:

```
val probe = TestProbe()
ingestService ! Retrieve(url, "myID", probe.ref)
val replies = probe.receiveWhile(1.second) {
    // this will only match for the right ID
    // and the full Record is bound to the variable `r`
    case r @ Record("myID", _) => r
    // EOF is not handled and will terminate this loop
  }
probe.expectMsg(0.seconds, EOF) // series must be finished already
```

Instead of following this with an expectNoMsg() call to verify that nothing arrives after the EOF message we might just append a second query. During testing we can ensure that there is only one instance active for this service which means that as soon as we receive the elements of the second response series we can be sure that the first one is properly terminated.

### 11.3.4 Providing Synchronous Execution Engines

The role of timeouts within those tests that are not timing-sensitive is only to bound the waiting time for a response that is expected. If we could arrange for the service under test to be executed synchronously instead of asynchronously then this waiting time would be zero: if the response is not ready when the method returns then it will also not become available at a later time since no asynchronous processing facilities are there to enable this.

Such configurability of the execution mechanism is not always available: synchronous execution can only be successful if the computation does not require intrinsic parallelism, it works best for those processes that are deterministic as discussed in chapter three, “Non-Determinism by Need”. If a computation is composed from Futures in a fully non-blocking fashion then this criterion is satisfied. Depending on the platform that is used there may be several ways to remove asynchrony during tests. Some

implementations, like Scala's Future, are built on the notion of an ExecutionContext which describes how the execution is realized for all tasks involved in the processing and chaining of Futures. In this case the only preparation that is necessary is to allow the service to be configured with an ExecutionContext from the outside, either when it is constructed or for each single request. Then the test procedure can pass one in that implements a synchronous event loop. Going back to the translation service this might look like the following:

```
val input = "Hur mår du?"
val output = "How are you?"
val ec = SynchronousEventLoop
val future = translate(input, ec)
future.value.get should be(Success(output))
```

For implementations which do not allow the execution mechanism to be configured in this fashion we can achieve the same effect by making the result container itself configurable. Instead of fixing the return type of the translate method to be a Future we could abstract over this aspect and allow any composable container to be passed in[11]. Future composition uses methods like map/flatMap/filter (Scala Future), then (JavaScript) or thenAccept (Java CompletionStage) and the only source code change that we need is to configure the service to use a specific factory for creating Futures so that we can inject one that performs computations synchronously.

---

Footnote 11 In other words what we are doing here is to abstract over the particular kind of monad that is used to sequence and compose the computation, allowing the test procedure to substitute the Future monad by the identity monad. In dynamically typed languages it is sufficient to create the monad's unit() and bind() functions, whereas in statically typed languages extra care needs to be taken to express the higher-kinded type signature of the resulting translate() method.

---

TODO: example in JavaScript

When it comes to other message-based components chances are not as good to find a way to make an asynchronous implementation synchronous during tests. One example is the Akka implementation of the Actor Model, which allows the execution of each Actor to be configured by way of selecting a suitable dispatcher. For test purposes there exists a CallingThreadDispatcher which will process each message directly within the context that uses the tell operator. If all Actors that contribute to the function of a given service are using only this dispatcher then sending a request will synchronously execute the whole processing chain such that possible replies are already delivered when the tell operator invocation returns. We can use this as follows:

```
val translationService = system.actorOf(
      Props[TranslationServiceActor].withDispatcher(
        "akka.test.calling-thread-dispatcher"))
val input = "Hur mår du?"
```

```
val output = "How are you?"
val probe = TestProbe()
translationService ! Translate(input, probe.ref)
probe.expectMsg(0.seconds, output) // assert immediately
```

The important change is that the Props describing the translation service Actor are configured with a dispatcher setting instead of leaving the decision to the ActorSystem. This needs to be done for each Actor that participates in this test case, meaning that if the translation service creates more Actors internally then it must be set up to propagate the dispatcher configuration setting to these (and they to their child Actors and so on; see the source code archives for details). It should be noted that this also requires several other assumptions:

- The translation service cannot make use of the system's scheduler since that would invoke the Actors asynchronously, potentially leading to the output not being transmitted to the probe when we expect it to.
- The same holds for interactions with remote systems, since those are by their very nature asynchronous.
- Failures and restarts would in this case also lead to asynchronous behavior since the translation service's supervisor is the system guardian which cannot be configured to run on the CallingThreadDispatcher.
- None of the Actors involved are allowed to perform blocking operations that might depend on other Actors running on the CallingThreadDispatcher since that would lead to deadlocks.

The list of assumptions could be continued with minor ones, but it should be clear that the nature of the Actor Model is at odds with synchronous communication, it relies upon asynchrony and unbounded concurrency. For simple tests—especially those which verify a single Actor—it can be beneficial to go this route, while higher-level integration tests involving the interplay of multiple Actors will usually require asynchronous execution.

So far we have discussed two widely used messaging abstractions, Futures and Actors, and each of them provides the necessary facilities to do synchronous testing if needed. Due to the ubiquity of this form of verification we will likely continue to see this support in all widespread asynchronous messaging abstractions, although there are already environments which are heavily biased against synchronous waiting—for example event-based systems like JavaScript—and which will drive the transition towards fully asynchronous testing. We embark on this spiritual journey in the following sections.

### 11.3.5 Asynchronous Assertions

The first step towards asynchronous testing is the ability to formulate an assertion that shall hold at a future point in time. In a sense we have seen a special case of this already in the form of TestProbe.expectMsg(). This method asserts that within a time interval from now on a message shall be received that has the given characteristics. A generalization of this mechanism is to allow arbitrary assertions to be used. ScalaTest offers this through its eventually() keyword. Using this we can rewrite our translation service test case:

```
val input = "Hur mår du?"
val output = "How are you?"
val future = translate(input)
eventually {
  future.value.get should be(Success(output))
}
```

This uses an implicitly supplied PatienceConfiguration that describes the time parameters of how frequently and for how long the enclosed verification is attempted before test failure is signaled. With this helper the test procedure itself remains fully synchronous but we obtain more freedom in expressing the conditions under which it shall proceed.

### 11.3.6 Fully Asynchronous Tests

We have found ways to express test cases for reactive systems within the framework of traditional synchronous verification procedures and most systems to date are tested in this fashion. But it feels wrong to apply a different set of tools and principles in the production and verification code bases, there is an impedance mismatch between these two that should be avoidable.

The first step towards fixing this was already made earlier in this chapter when we devised an Actor to verify the response latency characteristics of the EchoService. The ParallelSLATester is a fully reactive component that we developed to test a characteristic of another reactive component. The only incongruous piece in that test was the synchronous procedure used to start the test and await the result. What we would like to write instead is

```
val echo = echoService()
val N = 10000
val maxParallelism = 500
val controller = system.actorOf(Props[ParallelSLATester],
                                "keepSLAparallelController")
val future = controller ? TestSLA(echo, N, maxParallelism, _)
for (SLAResponse(timings, outstanding) <- future) yield {
  // discard top 5%
  val sorted = result.timings.sorted
```

```
  val ninetyfifthPercentile = sorted.dropRight(N * 5 / 100).last
  ninetyfifthPercentile should be < 2.milliseconds
}
```

Here we initiate the test by sending the TestSLA command to the Actor using the ask pattern to get back a Future for the later reply. We then transform that Future to perform the calculation and verification of the latency profile, resulting in a Future that will be either successful or failed depending on the outcome of the assertion in the next-to-last line. In traditional testing frameworks this Future will not be inspected, making this approach futile. An asynchronous testing framework on the other hand will react to the completion of this Future in order to determine whether the test was successful or not.

Combining such a test framework with the async/await extension available for .Net languages or Scala makes it straight-forward and easily readable to write fully asynchronous test cases. Our running example of the translation service would look like this:

```
async {
  val input = "Hur mår du?"
  val output = "How are you?"
  await(translate(input).withTimeout(5.seconds)) should be(output)
}
```

This has exactly the same structure as the initial synchronous version, marking out the asynchronous piece with await() and wrapping the whole case in an async{} block. The advantage over the intermediate version which used the blocking Await.result() construct is that the testing framework can execute many such test cases concurrently, reducing the overall time needed for running the whole test suite. This also means that we can relax the timing constraints because a missing reply will not bind as many resources as in the synchronous case, the Future for the next step of the test procedure will just not be set in motion; the five seconds in this example will also not tick so heavily on the wall clock as other test cases can make progress while this one is waiting.

As mentioned earlier JavaScript is an environment that is heavily biased towards asynchronous processing, blocking test procedures as are common in other languages are simply not feasible in this model. As an example we can implement the translation service test using *Mocha* and *Chai Assertions for Promises:*

```
describe('Translator', function() {
  describe('#translate()', function() {
    it('should yield the correct result', function() {
      return tr.translate('Hur mår du?')
               .should.eventually.equal('How are you?');
    })
  })
});
```

The Mocha test runner will execute several test cases in parallel, each returning a Promise as in this case. The timeouts after which tests are assumed to be failed if they did not report back can be configured at each level (globally, per test suite or test case).

### TESTING SERVICE LEVEL AGREEMENTS

With test cases being written in an asynchronous fashion we can revisit the latency percentile verification from another angle. The framework could allow the user to describe the desired response characteristics in addition to the test code itself and then automatically verify those by running the code multiple times in parallel. Doing so sequentially would be prohibitively expensive in many cases—we would not voluntarily have tested 10,000 iterations of the EchoService in the sequential version—and as discussed it would also not be a realistic measurement.

Going back to the SLA test of the echo service, the test framework would replace the custom ParallelSLATester actor that was used to communicate with the service under test:

```
async {
  val echo = echoService()
  val gauge = new LatencyTestSupport(system)
  val latenciesFuture =
    gauge.measure(count = 10000, maxParallelism = 500) { i =>
      val message = s"test$i"
      SingleResult((echo ? (Request(message, _))), Response(message))
    }
  val latencies = await(latenciesFuture, 20.seconds)
  latencies.failureCount should be(0)
  latencies.quantile(0.99) should be < 10.milliseconds
}
```

This is possible because the interaction between the test and the service is of a specific kind: we are performing a load test for a request–reply protocol. In this case we only need a factory for request–reply pairs that we can use to generate the traffic as needed, and the shape of the traffic is controlled by the parameters to the measure() method. The asynchronous result of this measurement is an object that contains the actual collections of results and errors that the 10,000 individual tests produced. These data can then easily be analyzed in order to assert that the latency profile fulfills the service level requirements.

### 11.3.7 Asserting the Absence of Asynchronous Errors

The last component to testing asynchronous components is that not all interactions with these will occur with the test procedure itself. Imagine a protocol adapter that is mediating between two components that have not been developed together and therefore do not understand the same message formats. In our running example with the translation service we might have first a version of the API which is based on text string serialization:

```
case class TranslateV1(query: String, replyTo: ActorRef)
```

The languages to be used for input and output are encoded within the query string and the reply that is sent to the replyTo address will be just a String. This works for a proof of concept, but later we might want to replace the protocol with a more strictly typed and intuitive version two:

```
case class TranslateV2(phrase: String,
                       inputLanguage: String,
                       outputLanguage: String,
                       replyTo: ActorRef)

sealed trait TranslationResponseV2
case class TranslationV2(inputPhrase: String,
                         outputPhrase: String,
                         inputLanguage: Language,
                         outputLanguage: Language)
case class TranslationErrorV2(inputPhrase: String,
                              inputLanguage: Language,
                              outputLanguage: Language,
                              errorMessage: String)
```

This redesign allows more advanced features like automatic detection of the input language to be implemented. Unfortunately other teams have progressed with implementing a translator using the version one protocol already, so let us assume that the decision is made to bridge between this and new clients by adding an adapter that accepts requests made with version two of the protocol and serves the replies that are provided by a translation service speaking the version one protocol in the background.

For this adapter we will typically write integration tests, making sure that given a functioning version one back-end it correctly implements version two of the protocol. In order to save maintenance effort, we will also write dedicated tests that just concentrate on the transformation of requests and replies; this will save time when debugging failures, since we now can more easily associate them with either the adapter or the back-end service. A test procedure could look like this:

```
val v1 = TestProbe()
```

```
val v2 = system.actorOf(TranslationService.propsV2(v1.ref))
val client = TestProbe()

// initiate a request to the adapter
v2 ! TranslateV2("Hur mår du?", "sv", "en", client.ref)

// verify that the adapter asks the V1 service back-end
val req1 = v1.expectMsgType[TranslateV1]
req1.query should be("sv:en:Hur mår du?")

// initiate a reply
req1.replyTo ! "How are you?"

// verify that the adapter transforms it correctly
client.expectMsg(TranslationV2("Hur mår du?", "How are you?",
  "sv", "en"))

// now repeat for translation errors
v2 ! TranslateV2("Hur är läget?", "sv", "en", client.ref)
val req2 = v1.expectMsgType[TranslateV1]
// this implicitly verifies that no other communication happened
req2.query should be("sv:en:Hur är läget?")
req2.replyTo ! "error:cannot parse input 'Hur är läget?'"

client.expectMsg(TranslationErrorV2("Hur är läget?", "sv", "en",
  "cannot parse input 'Hur är läget?'"))

v1.expectNoMsg(3.seconds)
```

Here the test procedure drives both the client side and the back-end side, stubbing each of them out as a TestProbe. The only active component that is executed normally is the protocol adapter. This allows us to formulate assertions not only about how the client-side protocol is implemented, it allows us to control the internal interactions as well. One such assertions is shown in the last line where we require the adapter to not make gratuitous requests to the service it is wrapping. Another benefit is that we can inspect the queries that are sent—see both occurrences of the TranslateV1 type—and fail early and with a clear error message if those are incorrect. In an integration test we would see only overall failures in this case.

This approach works well for such a 1-to-1 adapter, but it can become tedious or brittle for components that converse more intensely or more diversely with different back-ends. There is a middle ground between integration testing and fully controlled interactions: we can stub out the back-end services such that they are still executed autonomously but in addition to their normal function they keep the test procedure appraised of unexpected behavior of the component under test. To keep things simple we demonstrate this on the translation service adapter again.

```
case object ExpectNominal
case object ExpectError
case class Unexpected(msg: Any)

class MockV1(reporter: ActorRef) extends Actor {
  def receive = initial
```

```
  override def unhandled(msg: Any) = {
    reporter ! Unexpected(msg)
  }

  val initial: Receive = {
    case ExpectNominal => context.become(expectingNominal)
    case ExpectError   => context.become(expectingError)
  }

  val expectingNominal: Receive = {
    case TranslateV1("sv:en:Hur mår du?", replyTo) =>
      replyTo ! "How are you?"
      context.become(initial)
  }

  val expectingError: Receive = {
    case TranslateV1(other, replyTo) =>
      replyTo ! s"error:cannot parse input '$other'"
      context.become(initial)
  }
}
```

This mock of a version one back-end will provide the expected responses during a test, but it will only do so at the appropriate points in time: the test procedure has to explicitly unlock each of the steps by sending either an ExpectNominal or ExpectError message. Using this, the above test procedure changes to:

```
val asyncErrors = TestProbe()
val v1 = system.actorOf(mockV1props(asyncErrors.ref))
val v2 = system.actorOf(propsV2(v1))
val client = TestProbe()

// initiate a request to the adapter
v1 ! ExpectNominal
v2 ! TranslateV2("Hur mår du?", "sv", "en", client.ref)

// verify that the adapter transforms it correctly
client.expectMsg(TranslationV2("Hur mår du?", "How are you?",
  "sv", "en"))

// non-blocking check for async errors
asyncErrors.expectNoMsg(0.seconds)

// now verify translation errors
v1 ! ExpectError
v2 ! TranslateV2("Hur är läget?", "sv", "en", client.ref)
client.expectMsg(TranslationErrorV2("Hur är läget?", "sv", "en",
  "cannot parse input 'sv:en:Hur är läget?'"))

// final check for async errors
asyncErrors.expectNoMsg(1.second)
```

The test procedure in this case still drives both the client side and the back-end, but the latter is more autonomous which allows the test to be written more concisely. The first verification of the absence of asynchronous errors is performed such that it does not

introduce additional latency, and its purpose is only to aid in debugging test failures in case an asynchronous error from the nominal test step does not subsequently lead to directly visible test failures but instead only bubbles up in the last line of the test.

## 11.4 Testing Non-Deterministic Systems

The previous section introduced the difficulties that arise from the asynchronous nature of reactive systems. This had several interesting consequences even though the process that we were testing was fully deterministic: given a certain stimulus the component will eventually respond with the correct answer—the translation of a given phrase should always yield the same result. We discussed in chapter three that in distributed systems determinism cannot always be achieved, and the main reasons are unreliable means of communication and inherent concurrency. Since distribution is integral to reactive system design we nevertheless need to be able to test components that exhibit genuine non-determinism. Such tests are harder to express because the order of execution is not specified as a sequential progression of logic and for a single test procedure different outcomes or state transitions are possible and permissible.

### 11.4.1 The Trouble with Execution Schedules

Anyone who has written tests that are based on a particular event occurring within a specified time has likely seen spurious failures, where a test succeeds most of the time, but fails on occasion. But what if the execution is correct, but the timings are different because of varying insertion orders into queues, or different values being returned based on what was requested in what order?

It is imperative that application developers must define all of the correct behaviors that can occur based on varying execution schedules. This can be difficult, because it implies that the variance is finite and knowable, and the larger a system is and with greater numbers of interactions, this can be difficult with respect to precision. An example of a tool that supports this kind of functionality is Apache JMeter[12], where you can use Logical Controllers to fire requests in varying orders and timings to see whether or not the responses received match expectations for system behavior. Logical Controllers have other useful features as well, including request modification, request repeating and more. By executing tests with tools such as JMeter, you will root out more logical inconsistencies in your reactive application than if you always rely on test being executed in one order and one timing.

Footnote 12 http://jmeter.apache.org/

### 11.4.2 Testing Distributed Components

With distributed systems, which reactive applications are by definition, there are some more difficult problems that must be considered. Foremost is the idea that a distributed interaction can succeed in one dimension while failing in another. For example, imagine a distributed system where data must be updated across four nodes, but something goes awry on one of the servers and it never responds with a successful update response before a timeout occurs. These are known as partial failures[13], where latency can increase and throughput can fall because interactions between the many services that comprise the application are unable to complete all tasks as illustrated in Figure 11.3.

Footnote 13 http://en.wikipedia.org/wiki/Fault_tolerance

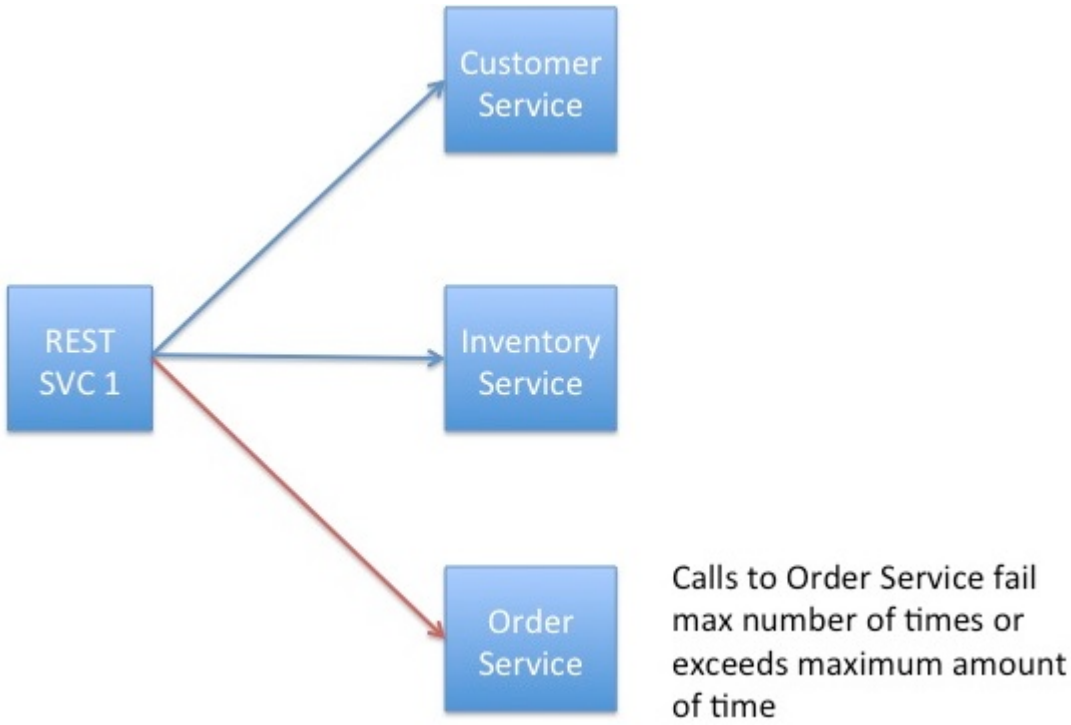


**Figure 11.3 Illustration of partial failure occurring when an interaction relies on three services and one interaction cannot be completed successfully**

What is particularly tricky about these kinds of failure is that it's unlikely that developers can consider all of the ways in which a reactive application may fail partially, and derive appropriate behavior for each case. Instead, developers should consider what the application should do when something occurs that they have not expected. We will discuss this in much more detail in Chapter 5: Fault Tolerance.

### 11.4.3 Mocking Actors

In order to show that a test passes or fails based on external interactions, a popular testing methodology is to mock or stub an external dependency. That means that when a class to be tested is constructed, the external services on which it depends are passed into the constructor of the class so that they are available at the time they are to be used. But mocking and stubbing that dependence class are two different approaches, each with their own tradeoffs:

#### MOCKS

Mocks are the concept of using an external library or framework to represent a fake instance of a class so that a developer can make assertions about whether or not a valid response can be properly handled. For example, imagine a class to be tested where it would attempt to persist the data in the class into a database. For unit tests, you would not want to actually test that interaction to the database, only that the class gives the appropriate result based on whether or not the attempt to perform that interaction was successful.

To create such tests, many mocking frameworks have sprung up in the past decade that allow you to create a "mock" instance of the class, where calls to the mocking framework instance can be preconfigured to return a specific value for that test. Examples of such frameworks on the Java platform include EasyMock, JMock, Mockito and ScalaMock. For each test, a mock instance is created and set up with the expected result to a particular interface, and when the call is made, that value is returned and permits the developer to make assertions that such behavior led to an appropriate result.

#### STUBS

Many developers consider the idea of using mocking frameworks to be an antipattern, where mocks do not represent an appropriate response mechanism for testing how such interactions take place. Such developer instead prefer to use stubs, or a test-only implementation of the interface that provides positive or negative responses for each interface based on the kind of response expected. This is considered less brittle at the time refactoring to an interface takes place, because the response from the stubbed method call is more well-defined.

There is a further consideration with respect to stubs that must be considered. Many developers complain that creating stubs for an interface is painful because it means that they have to provide implementations of each public interface even when they aren't used for a particular test, because the implementation of the interface specific to the test still requires each method in the interface to have defined behavior, even if that behavior is to do nothing. However, developers such as Robert Martin have argued that the usage of mocking frameworks is a "smell test" for APIs that have begun to exceed the Single

Responsibility Principle[14] - if the interface is painful to implement as a test stub because there are so many interfaces that have to be implemented, then the class interface is trying to do too much and is therefore exceeding the best practice rules for how each class should be defined with respect to the number of things it can do.

Footnote 14 http://en.wikipedia.org/wiki/Single_responsibility_principle

These kinds of arguments are best left to development teams to define, as it is easy to find edge cases that exceed such rules of thumb. As an extreme example, If you have an interface with a hundred public interfaces, you would have to provide an implementation of all of them in order to construct a stub for a single test, and this is difficult to do. It is even more difficult to maintain as the API changes during development. If your interfaces are small and represent atomic, granular responsibilities, writing stubs is much simpler, and allows for more expressive interpretations of how those dependencies can respond based on particular inputs, particularly when the interactions between the the calling class under test and the dependency become more complex.

#### REVERSE ONION TESTING PATTERN

A key concept for building effective tests for all applications is to create tests for the entire application from the inside and work outwards. This kind of testing is called the Reverse Onion pattern, where the approach is likened to peeling the layers of an onion inversely to how you would do so when actually cooking onions. This blends directly into the strategy of testing that was discussed in the beginning of the chapter. In taking this approach, the most minute expressions and functions are tested first, outwardly to services in isolation and then to the interactions themselves.

### *11.4.4 Distributed Components*

Having contextual handlers such as Akka TestKit's TestProbe are extremely handy for writing tests of this sort. Constructs like these allow you to differentiate between the responses for each request. Each test must have the ability to provide implementations, whether by mocks or stubs, of the actors/classes upon which they depend, so that you can enforce and verify the behavior you expect based on their responses. Once you have built tests with these characteristics, you can effectively test partial failure from the responses you get based on each failed interaction with each of those dependencies for an actor/class. If you are making a call to three services, and you want to test the behavior of the class when two of the three succeed and the third fails, you can do so by stubbing out behavior where the two successful stubs return expected values and the third returns an error (or never returns at all).

## 11.5 Testing Elasticity

For many developers, the concept of testing load or volume is well-known. However, for reactive applications, the focus changes from that traditional approaches to being able to verify that your application deployment is elastic within the confines of your available infrastructure. Just like everything else in application development, your system needs bounds in time and space, and the space limitation should be the maximum number of nodes you can spin up before you begin applying back pressure. In some cases you may be running on top of a Platform as a Service (Paas) where you "spill over" into public infrastructure like AWS or Rackspace images. But for many companies, that's not an option.

In order to test and verify elacity, you first have to know the bounds of throughput per node and the amount of infrastructure you will be deploying to. This should ideally come from the non-functional requirements of your application from the outset of the project, but if you have a clear idea of what your existing application's throughput profile looks like, you can start from there.

Assuming each node can handle 1,000 requests per second, and you have 10 nodes on which you can deploy, you want to test that traffic below a certain threshold only results in the minimum number of servers running that you specify. Tools such as Marathon with Mesos are run through Docker instances that you can query to see if nodes are up or down, and Marathon has a REST API through which you can make other assertions about the status of the cluster. For providing load to the system, there are several useful free utilities that do the job exceptionally well such as Bees With Machine Guns[15] and Gatling[16].

Footnote 15 https://github.com/newsapps/beeswithmachineguns

Footnote 16 http://gatling.io/

## 11.6 Testing Resilience

Application resilience is a term that needs to be deconstructed. Failure can occur at many levels, and each of them needs to be tested to ensure that an application can be reactive to virtually anything that can happen. The Reactive Manifesto states that "Resilience is achieved by replication, containment, isolation and delegation." This means that we need to be able to break down the varying ways that an application can fail, from the micro to the macro level, and test that it can withstand all of the things that can go wrong. In every case, a request must be handled and responded to regardless of what happened after it was received.

### 11.6.1 Application Resilience

First, there are the application concerns, where we must focus on behaviors specific to how the application was coded. These are the areas with which most developers are already familiar, and usually involves testing that an exception was received or what the application did as a result of injecting some data or functionality that should fail. In a reactive application, we should expect that exceptions or failures (as described in the callout in section 4.1 - How To Test) are not seen by the sender of a message, but communicated through other messages that elevate the failure to being domain events themselves.

This is an important point. Failure has traditionally been regarded as separate from the domain of the application, and is typically handled as tactical issues that must be prevented or communicated outside the realm of the domain for which the application was built. By making failure messages a first class citizen of the application domain itself, developers have the ability to handle failure in a much more strategic fashion. The developer can create two domains about the application: a domain of success and a domain of failure, and treat each appropriately with staged failure handling.

As an example, imagine an error retrieving valid data from an external source such as a database, where the call to retrieve the data succeeded but the data returned was not valid. This can be handled at one level of the application specific to that request, and either whatever valid data was retrieved is returned to the message sender, or a message connoting that the data was invalid is returned. However, if the connection to the external source is lost, that is a broader domain event than the individual request for the data, and should be handled by a higher-level component, such as whomever provided the connection that was used in the first place, so that it can begin the process of reestablishing the connection that was lost.

Application resilience comes in two forms, both external and internal. External resilience is handled through validation, where data passed into the application is checked to ensure it meets the requirement of the API, and if not, a notification is passed back to the sender (for example, a telephone country code that does not exist in a database of known numbers against which it could be checked). Internal resilience includes those errors that occur within the applications handling of that request once it has been validated.

## EXECUTION RESILIENCE

As discussed in previous chapters, the most important aspect to execution resilience is supervision of the thread or process where failure can occur. Without it, there is no way for you to discern what happened to a thread or process that fails, and in fact you may not have any way of knowing that a failure occurred at all. Once supervision is in place, you have the capability to handle failure, but you may not necessarily have the ability to test that you are doing the right thing as a result.

To get around this issue, developers sometimes expose internal state for the supervised functionality just so that they can effectively test whether that state was unharmed or somehow affected by the supervisor's management of it. For example, an actor that is resumed would see no change in its internal state, but an actor that was restarted would see its internal state return to the initial values it should have after construction. But this has a couple of problems:

1. How do you test an actor that should be stopped based on a specific kind of failure?
2. Is it a good idea to expose state that otherwise wouldn't be exposed just for verification purposes? This would represent a white box test, by the way.

In order to overcome these problems, there are a couple of patterns that can be implemented which give you the ability to determine what failure has occurred, or interact with a child actor that has test-specific supervision implemented. These are patterns that sound similar, but have different semantics.

It can be difficult to avoid implementing non-test-specific details inside of your tests. For example, if a test class were to attempt to create an actor directly from the ActorSystem as a child actor to the user guardian, the developer would not have control over how the supervision of errors that occur inside of that actor are handled. This may also be different than the expected behavior that is planned for the application, and will lead to invalid unit test behavior. Instead, a *StepParent* can be a test-only supervisor that creates an instance of the actor to be tested and delivers it back to the test client, who can then interact with it in any way they like. It merely exists to provide supervision external to the test class so that the test class is itself not the parent. Assuming we have a basic actor that we would like to test that can throw an Exception, it could look as simple as this:

```
class MyActor extends Actor {
  def receive = {
    case _ => throw new NullPointerException
  }
}
```

With that basic implementation, we can now create a StepParent strictly for the

purposes of testing which will create an instance of that actor from its own context, thus removing the test class itself from trying to fulfill that responsibility, like this:

```
class StepParent extends Actor {
   override val supervisorStrategy = OneForOneStrategy() {
     case thr => Restart
   }
   def receive = {
     case p: Props =>
       sender ! context.actorOf(p, "child")
   }
}
```

Now we can create a test that uses the StepParent to create our actor to be tested, and begin to test whatever behavior we want without having the supervision semantics in the test itself.

```
class StepParentSpec extends WordSpec
with Matchers with BeforeAndAfterAll {
  implicit val system = ActorSystem()

  "An actor that throws an exception" must {
    "Be created by a supervisor" in {
      val testProbe = TestProbe()
        val parent = system.actorOf(Props[StepParent], "stepParent")
        parent.tell(Props[MyActor], testProbe.ref)
        val child = testProbe.expectMsgType[ActorRef]
        … // Test whatever we want in the actor
    }
  }

  override def afterAll(): Unit = {
    system.shutdown()
  }
}
```

A *FailureParent* looks very similar, except that it also reports any failures it receives back to the testing class.  Assuming that we are going to test the same MyActor that we saw above, a FailureParent would look receive whomever it is supposed to report the failures back to as a constructor argument, and upon receipt of a failure, report it to that entity before performing whatever supervision work it intends to do, like this:

```
class FailureParent(failures: ActorRef) extends Actor {
  val props = Props[MyFailureParentActor]
  val child = context.actorOf(props, "child")

  override val supervisorStrategy = OneForOneStrategy() {
    case f => failures ! f; Stop
  }

  def receive = {
```

```
    case msg => child forward msg
  }
}
```

Now we can create a test that uses the StepParent to create our actor to be tested, and begin to test whatever behavior we want without having the supervision semantics in the test itself.

```
case object TestFailureParentMessage

class FailureParentSpec extends WordSpec
    with Matchers with BeforeAndAfterAll {
  implicit val system = ActorSystem()

  "Using a FailureParent" must {
    "Result in failures being collected and returned" in {
        val failures = TestProbe()
        val failureParent = system.actorOf(
        Props(new FailureParent(failures.ref)))
        failureParent ! TestFailureParentMessage
        failures.expectMsgType[NullPointerException]
    }
  }

  override def afterAll(): Unit = {
    system.shutdown()
  }
}
```

### API RESILIENCE

The previous examples of using StepParent and FailureParent are also a form of API resilience, where the messages being sent between actors are the API. In this way, you can think of actors as being very atomic examples of microservices. When requests are made of the service via its API, any data passed in must be validated to ensure it meets the contract of what the service expects. Once proven to be valid, the service can perform the work required to fulfill the request.

When building your own APIs, consider the impact of passing in a mechanism for failure so that you can verify through tests that the behavior of the service is correct. These can be called *Domain-Specific Failure Injectors*[17]. This can be done by either providing a constructor dependency that will simulate or produce the failure, or by passing it as part of the individual request. It may be entirely useful to create a class whose sole purpose is to randomize various kinds of failure so that they are tested at different times or execution orders to prove more thoroughly that the failure is appropriately handled. The Akka team has done this with their FailureInjectorTransportAdapter class for internal testing.

Footnote 17 http://en.wikipedia.org/wiki/Fault_injection

### 11.6.2 Infrastructure Resilience

Proving that your application is resilient is a great first step, but it is not enough. Applications depend on the infrastructure upon which they run, and they have no control over failures that can take place outside of themselves. Therefore, it is also important that anyone who is serious about implementing a reactive application also build or utilize a framework to help them test the application's ability to cope with infrastructure failures that happen around it.

Some may say the word partition and mean only from the network perspective, but that isn't necessarily true. Partitions happen any time a system has increased latency in response for any reason, including "stop the world" garbage collection, database latency, interminate looping, etc.

#### NETWORK RESILIENCE (LAN AND WAN)

One of the most notorious kinds of infrastructure failures is a *network partition*[18], where a network is incapable of routing between two or more subnetworks for various reasons. Networks can, and do, fail. Routers can go down just like any other computer, and occasionally paths provided by routing tables which are periodically revised and optimized cannot be resolved. It is best to assume that this will happen to your application and have a protocol for application management in the face of such an event.

Footnote 18 http://en.wikipedia.org/wiki/Network_partition

#### CLUSTER RESILIENCE

In the case of a network partition, it is entirely plausible that two or more nodes in a clustered application will not be able to reach each other, and each will assume leadership of a new subcluster that cannot be re-joined or merged. This is called the *Split-Brain problem*[19]. The optimistic approach is to allow the two or more subclusters to continue as normal, but if there is any state to be shared between them, this can be difficult to maintain as far as updates that occur in each being resolved to the correct final answer if and when they rejoin. The pessimistic approach is to assume all is lost and shut down both subclusters and attempt to restart the application entirely, so that consistency is maintained.

Footnote 19 http://en.wikipedia.org/wiki/Split-brain_(computing)

Some cluster management tools attempt to take a middling approach, where any subcluster with a majority of nodes (greater than 50% of known nodes) will automatically attempt to become the leader of a cluster that stays in operation. Any subcluster with less than 50% of known nodes will then automatically shut down. In theory, this sounds very reasonable, but cluster splits can be very unreasonable occurrences. It is entirely likely that such a split in the cluster will result in multiple

subclusters with less than 50% of known nodes in all of them, and they all shut down as a result. The operations teams that manage distributed systems in production have to be always on guard for such events, and again, have a protocol in place for handling them.

### NODE AND MACHINE RESILIENCE

Nodes, or instances, are processes running on a machine that represent one instance of an application currently able to perform work. If there is only one node in the entire application, it is not a distributed application and it represents a *single point of failure* (SPOF). If there is more than one node, possibly running on the same physical machine or across several of them, it is a distributed application. If all nodes are running on just one machine, this represents another SPOF, as any failure to the machine is going to take down the entire application.

To make an application provably reactive, you must be able to test that the removal of any node or machine at runtime does not affect your ability to withstand your application's expected traffic.

### DATA CENTER RESILIENCE

Similar to the other infrastructural concepts, deploying an application to a single data center is a SPOF and not a reactive approach. Instead, deployment to multiple data centers is a requirement to ensure that any major outage in one leaves your application with the capacity to handle all requests in others.

**SIDEBAR**

Netflix has created a suite of tools to help them test the robustness of their applications while running in production, called the Simian Army[20]. Netflix has had major outages happen in production and prefers to continue testing their application's resiliency at the node and machine level even in production. This gives them tremendous confidence that they can continue to service their customers even in the face of significant failures.

Footnote 20

http://techblog.netflix.com/2011/07/netflix-simian-army.html

To test node resilience, Netflix uses Chaos Monkey, which randomly disables production instances when executed. Note that this tool is only executed with operations engineers in attendance and closely monitoring for any outages that could occur as a result of the outages the tool induces. As a result of their success with this tool, Netflix created a legion of other such tools to check for latency, security credential expiration, unused resources and more.

To check resilience of an entire AWS availability zone, which is an isolation barrier within a deployment region, Netflix uses the Chaos Gorilla. This simulates a failure of an entire availability zone and checks whether their application is able to transition work to instances in other availability zones without downtime. To test data center resilience, they use the Chaos Kong tool, as they currently utilize multiple AWS Regions for the United States alone.

Whether or not you leverage existing tools, such as those from Netflix, or build your own, it is critical that you test your application's resilience in the face of myriad infrastructure failures to ensure that your users continue to get the responses they expect. Focus on applying these tools for any application that is critical to the success of your business.

## *11.7 Testing Responsiveness*

When testing elasticity and resilience, the focus is primarily in the number of requests your application can handle at any given time and with any given conditions. However, responsiveness is mostly about latency, or the time it takes to handle each request. As discussed in previous chapters, one of the biggest mistakes developers make is tracking latency by a metric defined qualitatively, typically by average. However, average is a terrible way of tracking latency, because it does not accurately reflect the variance in latency that your application is likely experience.

Instead, a latency target profile must be created for the application in order for the designers to understand how to assemble the system. For varying throughputs, the

expectation must be defined from the outset of what latency is acceptable at specific percentiles. Such a profile might look like the example given in Table 11.2.

**Table 11.2 Example of expected latency percentiles in relation to external load**

| Requests/s | 99% | 99.9% | 99.99% | 99.999% |
|---|---|---|---|---|
| 1,000 | 10ms | 10ms | 20ms | 50ms |
| 5,000 | 20ms | 20ms | 50ms | 100ms |
| 20,000 | 50ms | 50ms | 100ms | 250ms |

What is critical about this profile is that it clearly shows the expectation of how well the application should be able to respond to increased load, without falling over. Developers need to create tests that will verify that the response time is mapped appropriately against each percentile for each level of throughput, and this must be part of the continuous integration process to ensure that commits do not impact latency too negatively. There are free tools, such as HdrHistogram[21], which can help with the collection of this data and display it in a meaningful way.

Footnote 21 https://github.com/HdrHistogram/HdrHistogram

## 11.8 Summary

Testing and proving the ability to respond to varying loads, events and failures is a critical component to building a reactive application. Allow the tests guide the design by making choices based upon the results you see. At this point, you should have a clear understanding that:

- Testing must begin at the outset of the project and continue throughout every phase of its lifecycle
- Testing must be functional and non-functional to prove that an application is reactive
- Developers must write tests from the inside of the application outward to cover all interactions and verify correctness
- Elasticity is tested externally, while Resilience is tested in both infrastructure and internal components of the application

# 12 Fault Tolerance and Recovery Patterns

In this chapter you will learn how to incorporate the possibility of failure into the design of your application. We will demonstrate the patterns on the concrete use-case of building a resilient computation engine that allows batch job submission and their execution on elastically provisioned hardware resources. We build upon what we learnt in chapters 6 and 7, so you might want to refresh your understanding of those.

We start out with considering a single component and its failure and recovery strategies, then we build up more complex systems by hierarchical composition as well as client–server relationships. In order to achieve resilience against complete data loss at a single location we then introduce several different classes of state replication. In particular we discuss the following patterns:

- the Simple Component Pattern (a.k.a. the Single Responsibility Principle)
- the Error Kernel Pattern
- the Let-It-Crash Pattern
- the Circuit Breaker Pattern
- the Active–Passive Replication Pattern
- the Multiple-Master Replication Pattern
- the Active–Active Replication Pattern

These patterns are presented by first introducing their essence in one short paragraph (for easy reference when revisiting) followed by information on where the pattern emerged and then going into the details of an example setting where the pattern is applied to a concrete problem. Each pattern is then summarized with the concerns it addresses, its quintessential features and its scope of applicability.

## 12.1 The Simple Component Pattern

*«A component shall do only one thing, but do it in full.»*

This pattern applies wherever a system performs multiple functions or the functions it performs are so complex that they need to be broken up into different components. An

example is a text editor that includes spell checking: these are two separate functions (editing can be done without spell checking, and spelling can also be checked on the finished text and does not require editing capabilities), and on the other hand neither of these functions is trivial.

This pattern derives from the *Single Responsibility Principle* that was formulated by De Marco in his 1979 book on «Structured analysis and system specification» (Yourdon, New York). In its abstract form it demands to "maximize cohesion and minimize coupling", applied to object-oriented software design it is usually stated as "a class should have only one reason to change."

From the discussion of *divide et regna* in chapter 6 we know that in order to break a large problem up into a set of smaller ones we find help and orientation by looking at the responsibilities that the resulting components will have. Applying the process of responsibility division recursively allows us to reach any desired granularity and results in a component hierarchy that we can then implement.

### 12.1.1 The Problem Setting

As an example consider a service that offers computing capacity in a batch-like fashion: users submit jobs to be processed, stating a job's resource requirements and including an executable description of the data sources and the computation that is to be performed. The service will have to watch over the resources that it manages, it will have to implement quotas for the resource consumption of its clients and schedule jobs in a fair fashion. It will also have to persistently queue the jobs that it accepts such that clients can rely upon their eventual execution.

### 12.1.2 Applying the Pattern

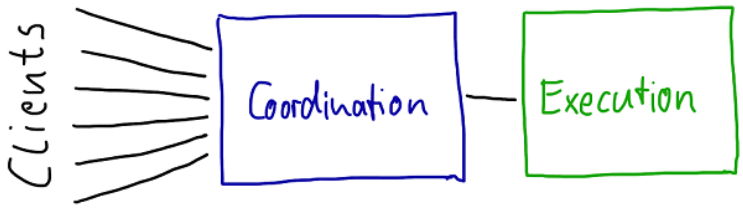


**Figure 12.1 Initial component separation**

One separation that we can immediately conclude is that the service implementation will be made up of two parts, one that does the coordination and that the clients communicate with and another that will be responsible for the actual execution of the jobs; this is shown in figure 12.1. In order to make the whole service elastic, the coordinating part would tap into an external pool of resources and dynamically spin up or down executor instances. We can see that the coordination will be a rather complex task and therefore we want to break it up further.

Following the flow of a single job request through this system we start with the job submission interface that is offered to clients. This part of the system needs to present a network endpoint that clients can contact, it needs to implement a network protocol for this purpose and it will interact with the rest of the system on behalf of the clients. We could break up responsibility even finer along these lines, but for now let us consider this aspect of representing clients within the client as one responsibility, the client interface will thus be our second dedicated component.

Once a job has been accepted and the client has been informed by way of an acknowledgement message our system must ensure that eventually the job will be executed. This can only be achieved by storing the incoming jobs on some persistent medium and we might be tempted to place this storage within the client interface component. But we can already anticipate that other parts of the system will have to access these jobs, for example in order to start their execution. This means that in addition to representing the clients this component would assume responsibility for making job descriptions accessible to the rest of the system, which is in violation of the single responsibility principle.

Another temptation might be to share the responsibility for the handling of job descriptions between the interested parties—at least the client interface and the job executor as we may surmise—but that will also greatly complicate each of these components as they will now have to coordinate their actions, running counter to the Simple Component Pattern's goal. It is much simpler to keep one responsibility within one component and avoid the communication and coordination overhead that comes with distributing it across multiple components. Besides these runtime concerns we also need to consider the implementation: sharing the responsibility means that one component needs to know about the inner workings of the other, their development needs to be tightly coordinated as well. Those are the reasons behind the second part of «do only one thing, *but do it in full*.»

This leads us to identify the storage of job descriptions as another segregated responsibility of the system and thereby as the third dedicated component. A valid interjection at this point is that the client interface component might well benefit from persisting the incoming jobs within its own responsibility, this would allow shorter response times for the job submission acknowledgement and it also makes the client interface independent from the job storage component in case of temporary unavailability. However, such a persistent queue would only have the purpose of eventually delivering accepted jobs to the storage component, who then will take responsibility of them. Therefore these notions are not in conflict with each other, we might implement both if system requirements demand it.

Taking stock, by now we have identified the client interface, the job storage, and the job executor as three dedicated components with non-overlapping responsibilities. What

remains to be done is to figure out which jobs to run in what order, we call this part “job scheduling”. The current state of our system’s decomposition is shown below, now we apply this pattern recursively until the problem is broken up into simple components.

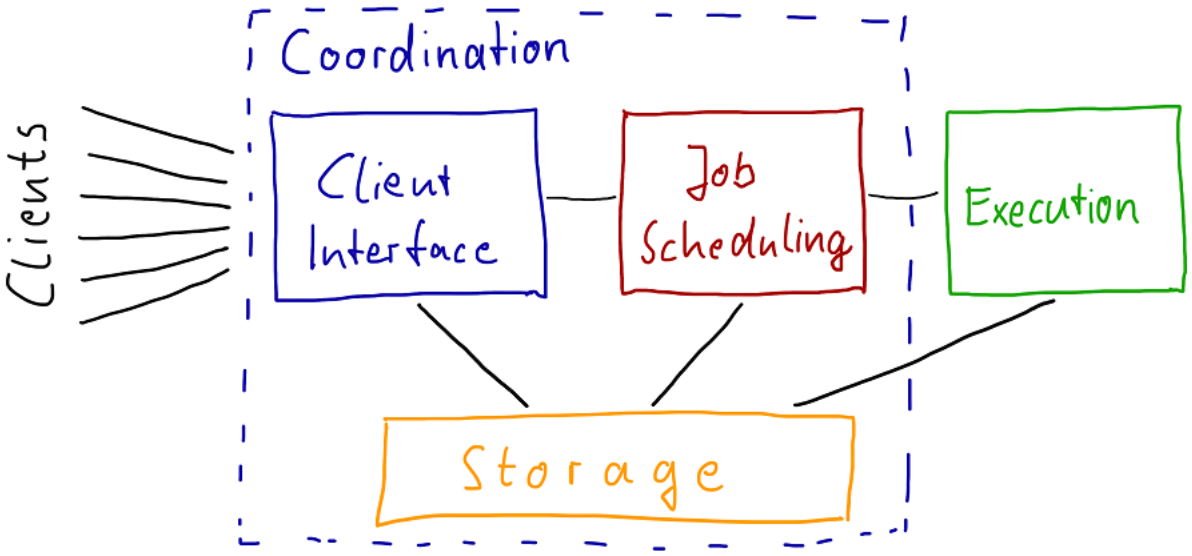


**Figure 12.2 Intermediate component separation with the Coordination component being broken up in three distinct responsibilities.**

Probably the most complex task in the whole service is to figure out the execution schedule for the accepted jobs, in particular when prioritization or fairness is to be implemented between different clients that share a common pool of resources—the corresponding allocation of computing shares are usually a matter of intense discussion between competing groups of people[104]. The scheduling algorithm will need to have access to job descriptions in order to extract scheduling requirements (maximum running time, possible expiry deadline, which kind of resources are needed, etc.), so this is another client of the job storage component.

Footnote 104 The authors have some experience with such allocation between different groups of scientists competing for data analysis resources in order to extract the insights they need for academic publications.

It takes a lot of effort—both for the implementation and at runtime—to plan the execution order of those jobs that are accepted for execution and this task is independent from deciding which jobs to accept. Therefore it will be beneficial to separate the responsibility of job validation into its own component. This also has the advantage of removing the rejected tasks before they become a burden for the scheduling algorithm. The overall responsibility of job scheduling now consists of two components, but its overall function should still be represented consistently to the rest of the system, for example the executors need to be able to retrieve the next job to run at any given time independently of whether there is a scheduling run in progress or not. For this reason we place the external interactions in an overall scheduling component of which the

validation and planning responsibilities are delegated to sub-components. The resulting split of responsibilities for the whole system is shown in the following diagram.

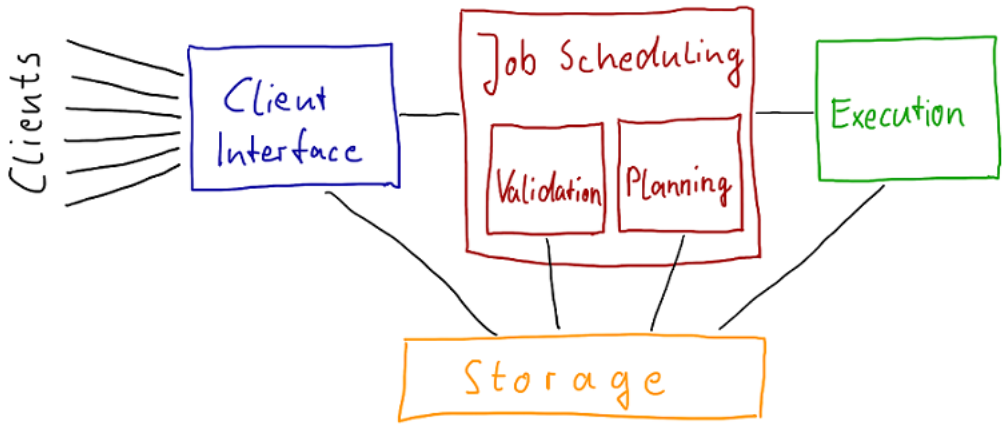


**Figure 12.3 The resulting component separation**

### 12.1.3 The Pattern Revisited

The goal of this pattern is to implement the Single Responsibility Principle and we did that by considering the responsibilities of the overall system at the highest level—client interface, storage, scheduling, execution—and separating these into dedicated components, keeping an eye on their anticipated communication needs. We then exemplarily dived into the scheduling component and repeated the process, finding that there are sizable and non-overlapping sub-responsibilities which we split out into their own sub-components. This left the overall scheduling responsibility in a parent component because we anticipate coordination tasks that will be needed independently of the sub-components' function.

By this process we arrived at segregated components that can be treated independently during the further development of the system. Each of these has one clearly defined purpose and each core responsibility of the system lies with exactly one component. While the overall system and the internals of any component may be complex the Single Responsibility Principle yields the simplest division of components to further work on—it frees us from always having to consider the whole picture when working on smaller pieces. This is its quintessential feature: it addresses the concern of system complexity.

Additionally, following the Simple Component Pattern simplifies the treatment of failures as we will exploit in the following two patterns.

### 12.1.4 Applicability

This is the most basic pattern to follow and it is universally applicable. Its application may lead you to a fine-grained split of your problem or to the realization that you are dealing with only one single component—the important part is that afterwards you know *why* you chose your system structure as you did. It helps all later phases of the design and implementation to document and remember this because when questions come up later of where to place certain functionality in detail you can let yourself be guided by the simple question of «what is its purpose?» The answer will directly point you towards one of the responsibilities you identified, or it will send you back to the drawing board in case you forgot to consider it.

It is important to remember that this pattern is meant to be applied in a recursive fashion, making sure that none of the identified responsibilities remain too complex or high-level. One word of warning, though: once you start dividing up components hierarchically it is easy to get carried away and go too far—the goal are simple components that have a real responsibility, not trivial components without an individual reason to exist.

## 12.2 The Error Kernel Pattern

*«In a supervision hierarchy keep important application state or functionality near the root while delegating risky operations towards the leaves.»*

This pattern builds upon the Simple Component Pattern and is applicable wherever components of different failure probability and reliability requirements are combined into a larger system or application—some functions of the system must “never” go down while others are necessarily exposed to failure. Applying the Simple Component Pattern will frequently leave you in this position, hence it pays to familiarize yourself well with the Error Kernel.

This pattern has been established in Erlang programs for decades[105] and was one of the main reasons that inspired Jonas Bonér to implement an Actor framework—Akka—on the JVM. The name “AKKA” has originally been conceived as the palindrome of “Actor Kernel”, referring to this core design pattern.

---

Footnote 105 The Ericsson AXD301’s legendary reliability is attributed in part to this design pattern and its success popularized both the pattern and the Erlang language and runtime that were used in its implementation.

---

### 12.2.1 The Problem Setting

From the discussion of hierarchical failure handling in chapter 7 we know that each component of a reactive system is supervised by another component that is responsible for its lifecycle management. This implies that if the supervisor component fails then all its subordinates will be affected by the subsequent restart, resetting everything to a known good state and potentially losing intermediate updates. If the recovery of important pieces of state data is expensive then such a failure will lead to extensive service downtimes, a condition that reactive systems aim to minimize.

In the following we illustrate the consequences this has on the example developed in the previous sections. Remember that we split the batch job processing system into six responsibilities: client interface, storage, overall scheduling, validation, planning, and execution. Now we need to consider what happens when things fail.

### 12.2.2 Applying the Pattern

Since recovering from a component's failure implies the loss and subsequent recreation of its state, we shall look for opportunities to separate likely points of failure from the places where important and expensive data are kept. The same applies to pieces that provide services which shall be highly available, these should not be obstructed by frequent failure nor long recovery times. In the example we identified the following disparate responsibilities:

- communication with clients (accepting jobs and delivering their results)
- persistent storage of job descriptions and their status
- overall job scheduling responsibility
- validation of jobs against quotas or authorization requirements
- job schedule planning
- job execution

Each of these responsibilities benefits from being decoupled from the rest, for example the communication with clients should not be obstructed by a failure of the job scheduling logic, just as client-induced failures should not affect the currently running jobs. The same reasoning applies to the other pieces analogously. This is another reason in addition to the single responsibility principle for considering them as dedicated components as shown again below.

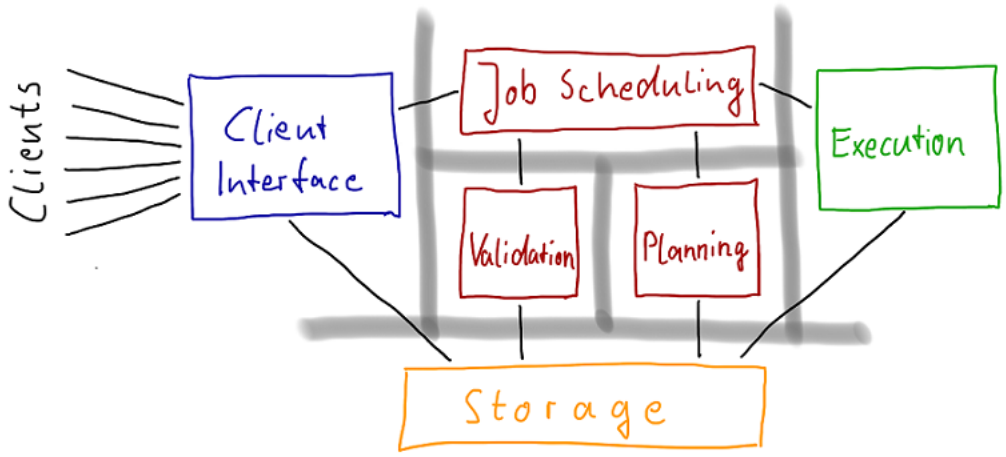


**Figure 12.4 The six components drawn as separate failure domains**

The next step is to consider the failure domains in the system and ask ourselves how each of them should recover and how costly that process will be. To this end we follow the path by which a job travels through the system.

Jobs enter the service through the communication component which speaks an appropriate protocol with the clients, maintaining protocol state and validating inputs. The state that is kept is short-lived, tied to the communication sessions that are currently open with clients. When this component fails affected clients will have to re-establish a session and possibly send commands or queries again, but our component does not need to take responsibility for these activities, in this sense it is effectively stateless—the state that it does keep is ephemeral and local. Recovery of such components is trivially done by just terminating the old and starting the new runtime instance.

Once a job has been received from a client it will need to be persisted, a responsibility that we placed with the storage component. This component will have to allow all other components to query the list of jobs, selecting them by current status or client account and holding all necessary meta-information. Apart from caches for more efficient operation this component does not hold any runtime state, its function is only to operate a persistent storage medium, therefore it can easily be restarted in case of failure. This assumes that the responsibility of providing persistence will be split out into a sub-component—which today is a likely approach—that we would have to consider as well: if the contents of the persistent storage becomes corrupted then it is a business decision whether to implement (partial) automatic resolution of these cases or leave it to the operations personnel; automatic recovery would presumably interfere with normal operation of the storage medium and would therefore fall into the storage component's responsibility.

The next stop of a job's journey through the batch service is the scheduling component. At the top level this one has the responsibilities of applying quotas and resource request validation as well as providing the executor component with a queue of

jobs to pick up. The latter is crucial for the operation of the overall batch service, without it the executors would run idle and the system would fail to perform its core function. For this reason we place this function at the top of the scheduling component's priorities and correspondingly at the root of its sub-component hierarchy as shown below.

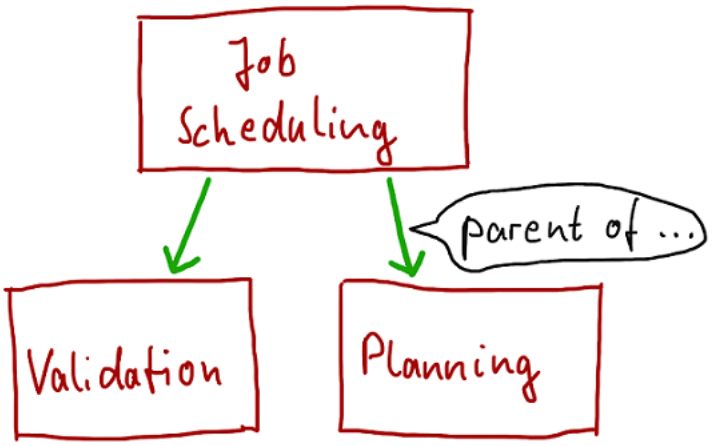


**Figure 12.5 Job scheduling sub-hierarchy**

While applying the Simple Component Pattern we identified two sub-responsibilities of the scheduling component. The first is to validate jobs against policy rules like per-client quotas[106] or general compatibility with the currently available resource set—it would not do to accept a job that needs 20 executor units when only 15 can be provisioned. Those jobs that pass validation form the input to the second sub-component that performs the job schedule planning for all currently outstanding and accepted jobs. Both of these responsibilities are task-based, they are started periodically and then either complete successfully or fail; failure modes include hardware failures as well as not terminating within a reasonable time frame. In order to compartmentalize possible failures these tasks should not directly modify the persistent state of jobs or the planned schedule but instead report back to their parent component who then takes action, be that notifying clients (via the client interface component) of jobs that failed their submission criteria or updating the internal queue of jobs to be picked next.

Footnote 106 For example one might want to limit the maximal number of jobs queued by one client—both in order to protect the scheduling algorithm and to enforce administrative limits.

While restarting the sub-components proved to be trivial restarting the parent scheduling component is more complex—it will need to initiate one successful schedule planning run before it can reliably resume performing its duties. Therefore we keep the important data and the vital functionality at the root and delegate the potentially risky tasks to the leaves. Here again we note that the Error Kernel Pattern confirms and reinforces the results of the Simple Component Pattern, we frequently find that the

boundaries of responsibilities and failure domains coincide and that their hierarchies match as well.

Once a job has reached the head of the scheduler's priority queue it will be picked up for execution as soon as computing resources become available. We have so far considered execution to be an atomic component, but considering failure we come to the conclusion that we will have to divide its function: the executor needs to keep track of which job is currently running where and it will also have to monitor the health and progress of all worker nodes. The worker nodes are those components that upon receiving a job description will interpret the contained information, contact data sources and run the analysis code that was specified by the client. Clearly the failure of each worker shall be contained to that node and not spread to other workers or the overall executor, which implies that the execution manager supervises all worker nodes as shown below.

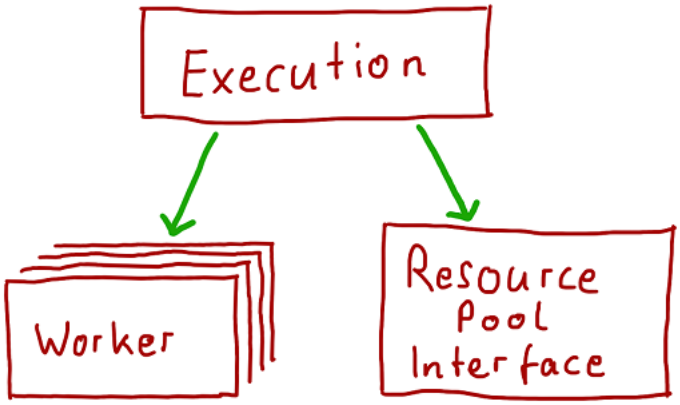


**Figure 12.6 Execution component sub-hierarchy**

If the system shall be elastic the executor will also make use of the external resource provision mechanism in order to create new worker nodes or shut down unused ones. The execution manager is also in the position to take the decision of enlarging or shrinking the worker pool because it naturally monitors the job throughput and it can easily be informed about the current job queue depth—another approach would be to let the scheduler decide the desired pool size. In any case it is the executor that holds the responsibility of starting, restarting or stopping worker nodes since it is the only component knowing when it is safe or required to do so.

Analogous to the client interface component the same reasoning applies in that the communication with the external resource provision mechanism should be isolated from the other activities of the execution manager, a communication failure in that regard should not keep jobs from being assigned to already running executor instances or job completion notifications from being processed.

The execution of the job was the main purpose of the whole service, but the journey of our job through the components is not yet complete. After the assigned worker node has informed the manager about the completion status this result needs to be sent back to the storage component in order to be persisted. If the job's nature was such that it must not be run twice, then the fact that the execution was about to start must also have been persisted in this fashion; in this case a restart of the execution manager will need to include a check of which jobs were already started but not yet completed prior to the crash and corresponding failure results will have to be generated. In addition to persisting the final job status the client will need to be informed about the job's result, which completes the whole process.

Now that we have illuminated the function and relationship of the different components we recognize that we have omitted one in the list of responsibilities above. The service itself needs to be orchestrated, composed from its parts, supervised and coordinated. We need one top-level component that creates the others and arranges for jobs and other messages being passed between them. In essence it is this component's function to oversee the message flow and thereby the business process of the service. This component will be top-level because of its integrating function that is needed at all times, even though it may be completely stateless by itself. The complete resulting hierarchy is shown below.

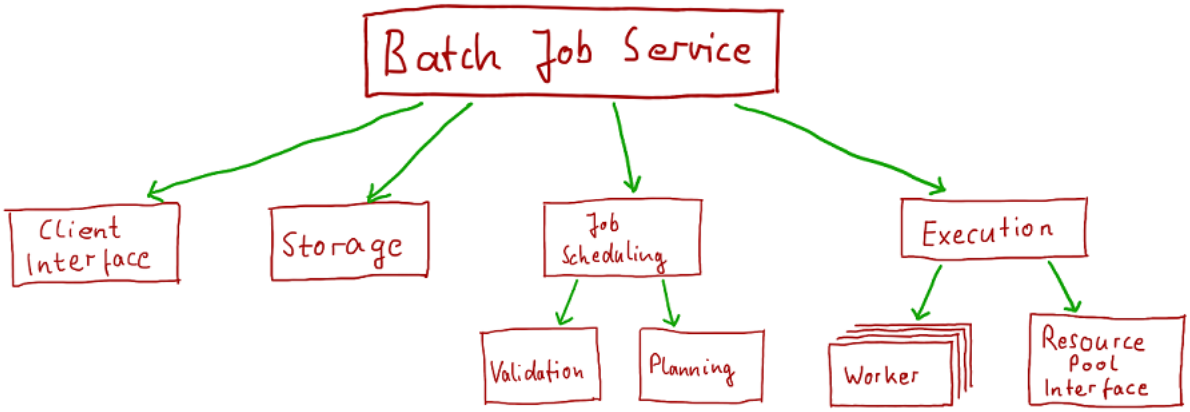


**Figure 12.7 The hierarchical decomposition of the batch job service**

### 12.2.3 The Pattern Revisited

The essence of what we did in the example above can be summarized in the following strategy: after applying the Simple Component Pattern pull important state or functionality towards the top of the component hierarchy, push activities that carry a higher risk for failure downwards towards the leaves. It is expected that the responsibility boundaries coincide with failure domains and that narrower sub-responsibilities will naturally fall towards the leaves of the hierarchy. This process may als lead you to introduce new supervising components that tie together the functionality of components that are otherwise siblings in the hierarchy, or it might guide you towards a more fine-grained component structure in order to simplify failure handling or decouple and isolate critical functions to keep them out of harm's way. The quintessential function of this pattern is to integrate the operational constraints of the system into its responsibility-based problem decomposition.

### 12.2.4 Applicability

The Error Kernel Pattern is applicable if any of the following are true:

- Does your system consist of components that have different reliability requirements?
- Do you expect components to have significantly different failure probabilities and failure severities?
- Does the system have important functionality that it must provide as reliably as possible while also having components that are exposed to failure?
- Is there important information kept in one part of the system that is expensive to recreate while other parts are expected to fail frequently?

The Error Kernel Pattern is not applicable if:

- no hierarchical supervision scheme is used
- the system is a Simple Component
- all components are either stateless or tolerant to data loss

We will discuss such scenarios in more depth later in this chapter when presenting the Active–Active Replication Pattern.

## 12.3 The Let-It-Crash Pattern

*«Prefer a full component restart to internal failure handling.»*

In chapter 7 we discussed "principled failure handling", noting that the internal recovery mechanisms of each component are limited because they are not sufficiently separated from the failing parts—everything within a component can be affected by a failure. This is especially clear for hardware failures that take down the component as a whole, but it is also true for corrupted state that is the result of some programming error that is only observable in rare circumstances. For this reason it is necessary to delegate

failure handling to a supervisor instead of attempting to solve it within the component itself.

This train of thought is also called *crash-only software*[107], the idea is that transient but rare failures are often very costly to diagnose and fix, making it preferable to recover a working system by rebooting parts of it. This way of hierarchical restart-based failure handling allows to greatly simplify the failure model and at the same time leads to a more robust system that even has a chance to survive failures that were entirely unforeseen.

Footnote 107 Both of the following articles are by George Candea and Armando Fox: "Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel", USENIX HotOS VIII, 2001 (http://dslab.epfl.ch/pubs/recursive_restartability.pdf) and "Crash-Only Software", USENIX HotOS IX, 2003 (https://www.usenix.org/legacy/events/hotos03/tech/full_papers/candea/candea.pdf)

### 12.3.1 The Problem Setting

We demonstrate this design philosophy on the example of the worker nodes that perform the bulk of the work in the batch service whose component hierarchy we developed in the previous two patterns. Each of these is presumably deployed on its own hardware—virtualized or not—that it does not share with other components, ideally there is no common failure mode between different worker nodes other than a complete computing center outage.

The problem that we are trying to solve is that the workers' code may contain programming errors that rarely manifest but when they do they will impede the ability to process batch jobs. Examples of this kind are very slow resource leaks that can go undetected for a long time but will eventually kill the machine; this could be open files, retained memory, background threads, etc. and the leak might not occur every time but could be caused by a rare coincidence of circumstances. Another example is a security vulnerability that allows the executed batch job to intentionally corrupt the state of the worker node in order to subvert its function and perform unauthorized actions within the service's private network—such subversion often is not completely invisible and leads to spurious failures that should better not be papered over.

### 12.3.2 Applying the Pattern

The Let-It-Crash Pattern by itself is very simple: whenever a worker node is detected to be faulty no attempts are made at repairing the damage, instead of doctoring with the internal state of the failed component we restart it completely, releasing all its resources and starting up again from scratch. If we obtained the worker nodes by asking an infrastructure service to provision them then we can go back to the most basic state imaginable, we decommission the old worker node and provision an entirely new one. This way no corruption or accumulated failure condition can have survived in the fresh instance since we start out from a known good state again.

This pattern can also be turned around so that components are "crashed" intentionally

on a regular basis instead of waiting for failures to occur. Deliberately inducing failures has been standard operation procedure for a long time in high-availability scenarios in order to verify that failover mechanisms are effective and perform according to their specification. The concept has been popularized in recent years by the “Chaos Monkey” employed by Netflix[108] to establish and maintain the resilience of their infrastructure. The chaotic nature of this approach manifests in that single nodes are killed at random without prior selection or human consideration, the idea is that in this way failure modes are exercised that could potentially be missed in human enumeration of all possible cases. On a higher level whole data centers or geographic regions are taken offline in a more prepared manner to verify global resource reallocation—this is done on the live production system since there is no simulation environment that could practically emulate the load and client dynamics in such a large-scale application.

Footnote 108 At the time of writing the largest streaming video provider in the U.S. The Chaos Monkey is part of the SimianArmy project which is available as open-source software at https://github.com/Netflix/SimianArmy, the approach is described in detail at http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html.

### 12.3.3 Implementation Considerations

While this pattern is deeply ingrained in reactive application design already it is nevertheless documented here to take note of its important consequences on the design of components and their interaction:

- Each component must tolerate a crash and restart at any point in time, just like a power outage can happen without warning. This means that all persistent state must be managed such that the service can resume processing requests with all necessary information and ideally without having to worry about state corruption.
- Each component must be strongly encapsulated so that failures are fully contained and cannot spread. The practical realization depends on the failure model for the hierarchy level under consideration, the options range from shared-memory message passing over separate O/S processes to separate hardware in possibly different geographic regions.
- All interactions between components must tolerate that peers can crash. This means ubiquitous use of timeouts and circuit breakers (described later in this chapter).
- All resources a component uses must be automatically reclaimable by performing a restart. Within an actor system this means that resources are freed by each actor upon termination or that they are leased from their parent; for an O/S process it means that the kernel will release all open file handles, network sockets, etc. when the process exits; for a virtual machine it means that the infrastructure resource manager will release all allocated memory (also persistent filesystems) and CPU resources, to be reused by a different virtual machine image.
- All requests sent to a component must be as self-describing as is practical so that processing can resume with as little recovery cost as possible after a restart.

### 12.3.4 Corollary: The Proactive Failure Signal Pattern

Let-it-crash describes how failures are dealt with, the other side of this coin is that failures must first be detected before they can be acted upon. In particularly catastrophic cases like hardware failures the supervising component can only detect that something is wrong by observing the absence of expected behavior, concretely this could mean that heartbeat requests go unanswered for too long. Applying this mechanism to all failure modes results in a high level of robustness already, but there are classes of failures where patiently counting out the suspected component takes longer than necessary: the component can diagnose some failures itself. A prominent example is that all exceptions that are thrown from an actor implementation will be treated as failures. These can be sent by the infrastructure (i.e. the actor library) to the supervisor in a message signaling the failure so that the supervisor can act upon it immediately. Wherever this is possible it should be viewed as an optimization of the supervisor's response time. The messaging pattern between supervisor and subordinate is depicted in figure 12.8 using the conventions established in appendix A.

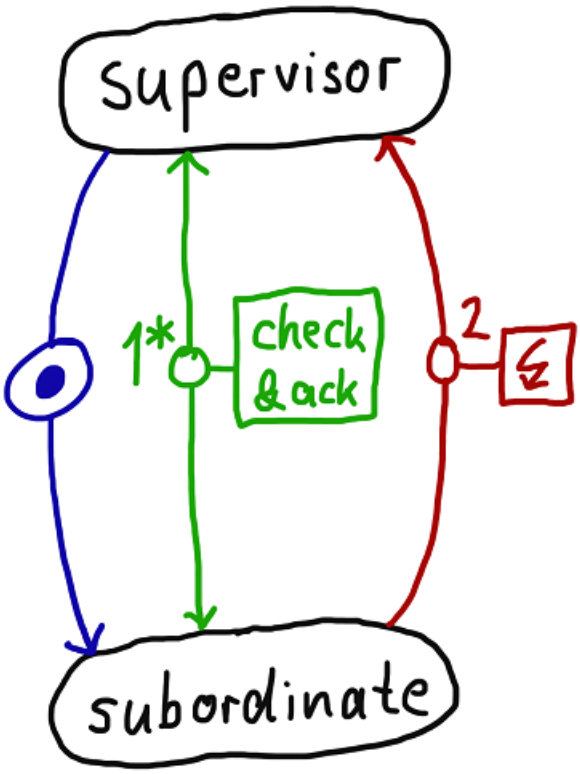


**Figure 12.8 The supervisor first starts the subordinate, then it performs periodic health checks by exchanging messages with it (step 1) until either no answer is returned or a failure signal is received (step 2).**

Depending on the failure model it can also be adequate to rely entirely on such measures. This is equivalent to saying that for example an actor is assumed to not have

failed until it has sent a failure signal. Monitoring the health of every single actor in a system is typically forbiddingly expensive and relying on these failure signals achieves sufficient robustness at the lower levels of the component hierarchy. In order to guard against unforeseen failures proper external health monitoring should be implemented on a higher level, though. In many cases this is possible by monitoring the service quality (failure rate, response latency, etc.) during normal operation in order to avoid the implementation of pure “ping” requests for all components within the hierarchy of the monitored service.

## 12.4 The Circuit Breaker Pattern

*«Protect services by breaking the connection to their users during prolonged failure conditions.»*

In the previous sections we have discussed how to segregate a system into a hierarchy of components and sub-components for the purpose of isolating responsibilities and encapsulating failure domains. This pattern describes how to safely connect different parts of the system together so that failures do not spread uncontrollably across them. Its origin lies in electrical engineering: in order to protect electrical circuits from each other and introduce decoupled failure domains we have established the technique of breaking the connection when the transmitted power exceeds a given threshold.

Translated to a reactive application this means that the flow of requests from one component to the next may be broken up deliberately when the recipient is overloaded or otherwise failing. This serves two purposes: firstly the recipient gets some breathing room to recover from possible load-induced failures and secondly the sender decides that requests will fail instead of wasting time with waiting for negative replies.

While circuit breakers are used in electrical engineering since the 1920’s, the use of this principle has been popularized in software design only recently, e.g. by Michael Nygard’s book «Release It!» (Pragmatic Programmers, 2007).

### 12.4.1 The Problem Setting

The batch job execution facility we designed in the previous two sections will serve us yet another time, we already hinted at one place that would do well to include a circuit breaker: the service is offered to external clients who submit jobs at their own rate and schedule and those are not naturally bounded by the capacity of the batch system.

To visualize what this means we consider a single client that contacts the batch service to submit a single job. The client will get multiple status updates as the submitted job progresses through the system:

- upon having received and persisted the job description
- upon having accepted the job for execution, or upon rejecting it due to policy violations
- upon starting execution

- upon finishing execution

The first of these steps is a very important one, it assures the client that there will be further updates on this job because it has been admitted into the system and will at least be examined. Providing this guarantee is quite costly—it involves storing the job description in non-volatile and replicated memory—and therefore a client could easily generate more jobs per second than the system can safely ingest. In this case the client interface would overload the storage subsystem, which has knock-on effects for the job scheduling and execution components who would now experience degraded performance when accessing job descriptions for their purposes. The system might still work in this state, but its performance characteristics would be quite different from normal operation, it would be in "overload mode". This is a condition that we should strive to avoid when possible because running at 100% capacity is in most cases less efficient than leaving a little bit of headroom. The reason for this is that at full capacity more time will be wasted competing for the available resources (CPU time, memory bandwidth, caches, IO channels) than otherwise when requests can travel through the system mostly unhindered by congestion.

There is a second problem to be considered here that may or may not be attributed to client-induced overload. The client interface cannot acknowledge reception of a job description before it receives the successful reply from the storage subsystem. If this reply does not arrive within the allotted time then the response to the client will be delayed for longer than the SLA allows—the service will violate its latency bound.

This means that during time periods where the storage subsystem fails to answer promptly the client interface will have to come to its own conclusions—if it cannot ask another (non-local) component then it must locally determine the appropriate response to its own clients.

### *12.4.2 Applying The Pattern*

When implementing the client interface module we will have to write one piece of code that sends requests to the storage subsystem. If we make sure that all such requests take this one route then we have an easy time reacting to the problematic scenarios outlined above. What we will have to do is to keep track of the response latency for all requests that are made. When we observe that this latency rises consistently above the agreed limit then we switch into "emergency mode": instead of trying new requests we answer all subsequent ones with negative replies right away—we fabricate the negative replies on behalf of the storage subsystem since that one cannot do even that within the allowed time window under the current conditions.

In addition we should also monitor the failure rate of replies that come back from the storage subsystem. It does not make much sense to send ever more storage requests when all of them will be answered negatively anyway; instead of wasting the network

bandwidth we should switch into “emergency mode” as well, fabricating these negative replies.

An example implementation of this scheme in Akka would look like the following:

```
private object StorageFailed extends RuntimeException

private def sendToStorage(job: Job): Future[StorageStatus] = {
  // make an asynchronous request to the storage subsystem
  val f: Future[StorageStatus] = …
  // map storage failures to Future failures to alert the breaker
  f.map {
    case StorageStatus.Failed => throw StorageFailed
    case other                => other
  }
}

private val breaker = CircuitBreaker(
    system.scheduler, // used for scheduling timeouts
    5, // number of failures in a row when it trips
    300.millis, // timeout for each service call
    30.seconds, // time before trying to close after tripping
  )

def persist(job: Job): Future[StorageStatus] =
  breaker
    .withCircuitBreaker(sendToStorage(job))
    .recover {
      case StorageFailed                  => StorageStatus.Failed
      case _: TimeoutException            => StorageStatus.Unknown
      case _: CircuitBreakerOpenException => StorageStatus.Failed
    }
```

The other client interface code will call the persist method and get back a Future representing the storage subsystem’s reply, but the remote service invocation will only be performed if the circuit breaker is in the *closed* state. Negative replies (StorageStatus.Failed) or timeouts will be counted by breaker and if it sees five failures in a row then it will transition into the *open* state in which it immediately provides a response that consists of a CircuitBreakerOpenException. After 30 seconds exactly one request will be let through again to the storage subsystem and if that comes back successfully and in time then the breaker flicks back into the *closed* state.

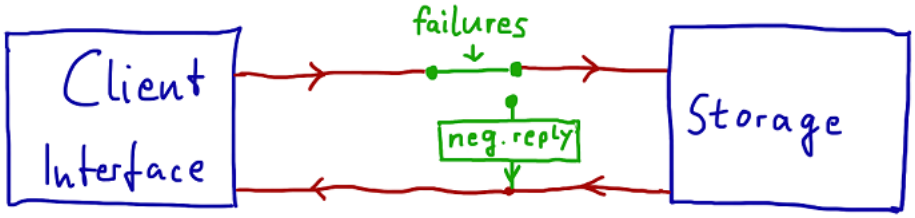


**Figure 12.9 A circuit breaker between the client interface and the storage subsystem**

What we have done so far means that the client interface will reply to the external clients within its allotted time, but in case of storage subsystem overload or failure these

replies will be fabricated and negative for all clients alike. While this protects the system from attacks it is not the best we can do. Just as in electrical engineering we need to break circuits at more than one level—what we have built so far is the main circuit breaker for the whole apartment house but we are lacking the power distribution boards that limit the damage that each individual tenant can do.

There is one difference between these per-client circuit breakers and the main one: they react not primarily to trouble downstream, they enforce a certain maximum current to flow through them. While in computer systems this might rather be called rate limiting this is exactly the function of circuit breakers in electrical engineering. What this means is that instead of tracking the call latencies we must remember the times of previous requests and reject new requests that violate a stated limit like “no more than 100 requests in any 2 sec interval”. Writing such a facility in Scala is quite straight-forward:

```
import scala.concurrent.duration.FiniteDuration
import scala.concurrent.duration.Deadline
import scala.concurrent.Future

case object RateLimitExceeded extends RuntimeException

class RateLimiter(requests: Int, period: FiniteDuration) {
  private val startTimes = {
    val onePeriodAgo = Deadline.now - period
    Array.fill(requests)(onePeriodAgo)
  }

  // the index of the next slot to be used, keeping track of when
  // the last job was enqueued in it to enforce the rate limit
  private var position = 0

  private def enqueue(time: Deadline) = {
    startTimes(position) = time
    position += 1
    if (position == requests) position = 0
  }

  def call[T](block: => Future[T]): Future[T] = {
    val now = Deadline.now // obtain current timestamp
    if ((now - startTimes(position)) < period) {
      Future.failed(RateLimitExceeded)
    } else {
      enqueue(now)
      block
    }
  }
}
```

Now we can combine both kinds of circuit breakers to obtain the full picture. For each client (which usually includes any request that uses certain authentication credentials—independent of which network connection the request arrived on) we maintain a RateLimiter and once within the client interface we use one shared CircuitBreaker to guard the remote invocations of the storage subsystem. The per-client code could look like the following:

```
private val limiter = new RateLimiter(100, 2.seconds)

// this is assumed to not be invoked concurrently as it is for a
// single client
def persistForThisClient(job: Job): Future[StorageStatus] =
  limiter
    .call(persist(job))
    .recover {
      case RateLimitExceeded => StorageStatus.Failed
    }
```

### ADVANCED USAGE

It is common practice to gate a client that repeatedly violates its rate limit, this is an incentive to client code writers to properly limit the service calls on their end instead of always sending at full speed—with the code above that would be an efficient tactic for achieving maximum throughput. In order to do that we only need to add another circuit breaker:

```
private val limiter = new RateLimiter(100, 2.seconds)
private val breaker = CircuitBreaker(system.scheduler,
                                     10, Duration.Zero, 10.seconds)

def persistForThisClient(job: Job): Future[StorageStatus] =
  breaker
    .withCircuitBreaker(limiter.call(persist(job)))
    .recover {
      case RateLimitExceeded             => StorageStatus.Failed
      case _: CircuitBreakerOpenException => StorageStatus.Gated
    }
```

In order to trip the circuit breaker the client will have to send ten requests while being above the rate limit; assuming regular request spacing this means that the client needs to submit at least at 10% higher rate than it is allowed. In this case it will be blocked from service for the next 10 seconds and it will be informed by way of receiving a Gated status reply. The Duration.Zero in the code above has the function of turning off the timeout tracking for individual request, this is not needed here since it will be performed by the persist call. The complete setup is shown in the following diagram:

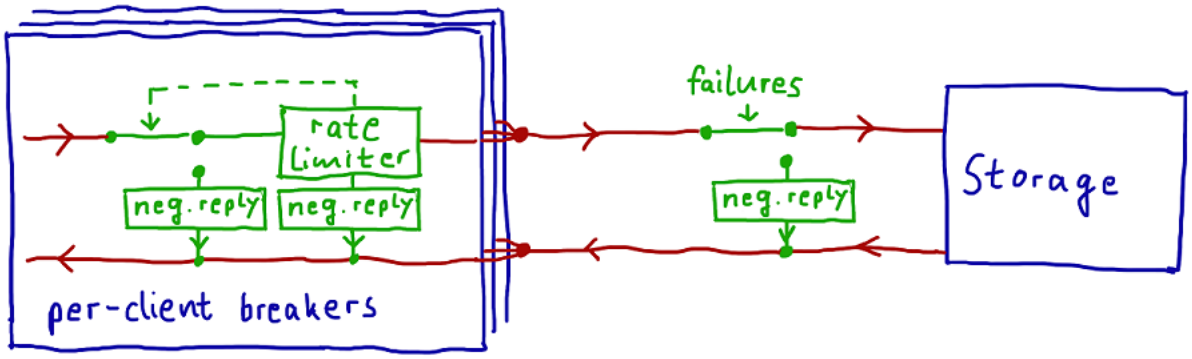


**Figure 12.10 Complete circuit breaker setup between client interface and storage subsystem**

### 12.4.3 The Pattern Revisited

We have decoupled the client interface and the storage subsystem by introducing predetermined breaking points on the path from one to the other. Thereby we have protected the storage from being overloaded in general (the main circuit breaker) and we have protected the client interface's function from single misbehaving clients (the rate limiting per-client circuit breakers); in the other direction we have also protected the client interface from failures of the storage subsystem, fabricating negative responses in case no others are readily available.

We have done this considering overload scenarios, but now that the code is in place to handle the situation it does not matter anymore why the storage subsystem does not answer successfully or in time, the reaction is independent of the underlying reason. This makes the system more resilient compared to handling every single error case separately. And this is what is meant by bulk-heading failure domains to achieve compartmentalization and encapsulation.

### 12.4.4 Applicability

This pattern is applicable wherever two decoupled components communicate and where failures shall not travel upstream to infect and slow down other components or where overload conditions shall not travel downstream to induce failure. Decoupling has a cost in that all calls pass through another tracking step and timeouts need to be scheduled, hence it should not be applied on a too small level of granularity, it is most useful between different services in a system. This applies especially to services that are reached via network connections, where the circuit breaker also reacts appropriately to network failures by concluding that the remote service is not currently reachable.

Another important use of circuit breakers is that monitoring their state reveals interesting insight into the runtime behavior and performance of a service. When circuit breakers that protect a given service are tripping the operations personnel will usually want to be alerted in order to look into the outage.

## 12.5 The Active–Passive Replication Pattern

*«Keep multiple copies of the service running in different locations, but only accept modifications to the state in one location at any given time.»*

This pattern is also sometimes referred to as "fail-over" or "master–slave replication". We have already seen one particular form of fail-over: the ability to restart a component means that after a failure a new instance is created and takes over the functionality, like passing the baton from one runner to the next. For a stateful service this means that the new instance will access the same persistent storage location as the previously failed one, recovering the state before the crash and continuing from there. This only works as long as the persistent storage is intact; if that fails then restarting is not possible, the service

will forget all its previous state and start from scratch. Imagine your web shop forgetting about all registered users, that would be a catastrophe!

We use the term *active–passive replication* to more precisely denote that in order to be able to recover even when one service instance fails completely—including loss of its persistent storage—we distribute its functionality and its full data set across several physical locations. The need for such measures was discussed in chapter 2 under the heading of *Reacting to Failure:* replication means to not put all eggs in one basket.

Replicating and thereby distributing a piece of functionality will require a certain amount of coordination, in particular considering operations that change the persistent state. The goal of active–passive replication is to ensure that at any given time only one of the replicas has the right to perform modifications. This allows individual modifications to be made without requiring consensus about them as long as there is consensus about who the currently active party is, just like electing a mayor serves the purpose of simplifying the process of coordination within a city.

*TODO: history and further reading*

### 12.5.1 The Problem Setting

In our batch service example the most important component that is used by all the others is the Storage module. Restarting this component after a failure will allow it to recover from many issues but we must take care to protect its data from being lost. This means storing the data in multiple places and allowing the other components to retrieve it from either of them.

In order to visualize what this means we will consider an incoming request that arrives at the Client Interface. Since the Storage component now is spread across multiple locations the Client Interface will need to know multiple addresses in order to talk to it. We assume that there is a service registry via which all components can obtain the addresses of their communication partners. When the Storage component starts up it will register all its locations and their addresses in the registry where the client interface will be able to retrieve them. This allows new replicas to be added or old ones to be replaced at runtime. While a static list of addresses might be sufficient in some cases it is in general desirable to be able to change addresses, especially in a cloud computing environment.

Which of the replicas is active one will change over time and there are several options for routing requests to it:

- The Storage component could inform its clients via the service registry of which address belongs to the currently active replica. This simplifies the implementation of the client but it also leads to additional lag for picking up a change of replica after a failure.
- The internal consensus mechanism for electing the active replica could be made accessible to the clients, allowing them to follow changes by listening in on the election protocol. This provides timely updates but couples the implementation of client and

service by requiring them to share a larger protocol.

- All replicas could offer the service of forwarding requests to the currently active replica. This frees the clients from having to track replica changes closely while avoiding downtimes or close coupling. A possible downside of this approach is that requests sent via different replicas may arrive with substantially different delays, thereby making the ordering of request processing less deterministic.

An important property of any implementation must be that once the service replies with a confirmation the request must have been processed and its results persisted such that after a subsequent failure the new active replica will behave accordingly.

### 12.5.2 Applying The Pattern

In the following we study an implementation of this replication scheme based upon Akka Cluster. This allows us to delegate the election of the active replica to the Cluster Singleton feature that is offered by this library. A cluster wide singleton is an actor that is spawned on the oldest cluster member with a given role. The Akka implementation ensures that there cannot be conflicting information on which the oldest member is within the same cluster which means that there can be two instances of the cluster singleton running simultaneously. This guarantee relies upon the proper configuration of the cluster itself: during a network partition each of the isolated parts will have to decide whether to continue operation or shut itself down and if the rules are formulated such that two parts can continue running then a singleton will be elected within each of these parts. Where this is not desired a strict quorum must be employed that is larger than half the total number of nodes in the cluster—with the consequence that during a three-way split the whole cluster may shut down. Further discussion of these topics can be found in Chapter 17, for now it is sufficient to know that the Cluster Singleton mechanism ensures that there will only be one active replica running at any time.

As a first step we implement the actor that controls the active replica. This actor will be instantiated by the cluster singleton mechanism as a cluster wide singleton as explained above. Its role is to accept and answer requests from clients as well as to disseminate updates to all passive replicas. To keep things simple we implement a generic key–value store that associates JSON values and uses text strings as keys. This saves us the trouble of defining the required data types for our batch service operation—which is not central to the application of this pattern in any case. The full source code for this example can be found in the github repository. Before we begin we introduce the protocol messages by which clients interact with the replicated storage:

```
case class Put(key: String, value: JsValue, replyTo: ActorRef)
case class PutConfirmed(key: String, value: JsValue)
case class PutRejected(key: String, value: JsValue)
case class Get(key: String, replyTo: ActorRef)
case class GetResult(key: String, value: Option[JsValue])
```

In response to a Put command we expect either the a confirmation or rejection reply whereas the result of a Get command will always indicate the currently bound value for the given key which may optionally be empty. A command may be rejected in case of replication failures or service overload as we will see below. The type JsValue represents an arbitrary JSON value in the play-json library but the choice of serialization library is not essential here.

When the singleton actors start up it will first have to contact a passive replica to obtain the current starting state. It will be most efficient to ask the replica within the same actor system i.e. on the same network host since this avoids serializing the whole data store and sending it over the network. In our implementation the address of the local replica is provided to the actor via its constructor:

```
class Active(localReplica: ActorRef,
             replicationFactor: Int,
             maxQueueSize: Int)
    extends Actor with Stash with ActorLogging {

  // the data store, held in memory sake of a simple example
  private var theStore: Map[String, JsValue] = _
  // sequence number generator for Replicate requests
  private var seqNr: Iterator[Int] = _

  // ask for InitialData to be provided by the local storage replica
  log.info("taking over from local replica")
  localReplica ! TakeOver(self)

  def receive = {
    case InitialState(m, s) =>
      log.info("took over at sequence {}", s)
      theStore = m
      seqNr = Iterator from s
      context.become(running)
      unstashAll()
    case _ => stash()
  }

  val running: Receive = ... // behavior of the running state
}
```

While the actor is waiting for the initial state message it will need to ignore all incoming requests. Instead of dropping them or making up fake replies we stash them within the actor, to be answered later as soon as we have the necessary data. Akka directly supports this usage by mixing in the Stash trait. In the running state the actor will make use of the data store and the sequence number generator but it will need several more data structures to organize its behavior:

```
class Active(localReplica: ActorRef,
             replicationFactor: Int,
             maxQueueSize: Int)
    extends Actor with Stash with ActorLogging {

  ... // initialization as shown above
```

```
  private val MaxOutstanding = maxQueueSize / 2

  private val toReplicate = Queue.empty[Replicate]
  private var replicating = TreeMap.empty[Int, (Replicate, Int)]

  import context._

  // recurring timer to resend outstanding replication requests
  val timer = system.scheduler.schedule(1.second, 1.second, self, Tick)
  override def postStop() = timer.cancel()

  val running: Receive = {
    case p @ Put(key, value, replyTo) =>
      if (toReplicate.size < MaxOutstanding) {
        toReplicate.enqueue(Replicate(seqNr.next, key, value, replyTo))
        replicate()
      } else {
        replyTo ! PutRejected(key, value)
      }
    case Get(key, replyTo) =>
      replyTo ! GetResult(key, theStore get key)
    case Tick =>
      replicating.valuesIterator foreach {
        case (replicate, count) => disseminate(replicate)
      }
    case Replicated(confirm) =>
      replicating.get(confirm) match {
        case None => // already removed
        case Some((rep, 1)) =>
          replicating -= confirm
          theStore += rep.key -> rep.value
          rep.replyTo ! PutConfirmed(rep.key, rep.value)
        case Some((rep, n)) =>
          replicating += confirm -> (rep, n - 1)
      }
      replicate()
  }

  /**
   * helper method that dispatches further replication requests when
   * appropriate
   */
  private def replicate(): Unit =
    if (replicating.size < MaxOutstanding && toReplicate.nonEmpty) {
      val r = toReplicate.dequeue()
      replicating += r.seq -> (r, replicationFactor)
      disseminate(r)
    }

  /**
   * send a replication request to all replicas, including local
   */
  private def disseminate(r: Replicate): Unit = {
    val req = r.copy(replyTo = self)
    def replicaOn(addr: Address): ActorSelection =
      actorSelection(localReplica.path.toStringWithAddress(addr))
    val members = Cluster(system).state.members
    members.foreach(m => replicaOn(m.address) ! req)
  }
}
```

First of all the actor keeps a queue of items to be replicated, called toReplicate, plus a queue of replication requests that are currently in flight. The latter—replicating—is implemented as an ordered map because we will need to have direct access to its elements as replication requests complete. Whenever the actor receives a Put request it

will check whether there is still room in the queue of items to be replicated. In case the queue is full the client is immediately informed that the request is rejected, otherwise a new Replicate object is enqueued that describes the update to be performed and then replicate() method is invoked. This method transfers updates from the toReplicate queue to the replicating queue if there is space. The purpose of this setup is to place a limit on the number of currently outstanding replication requests so that clients can be informed when the replication mechanism cannot keep up with their generated update load.

When an update is moved to the replicating queue the disseminate function is called. Here we implement the core piece of the algorithm: every updates that is accepted by the active replica will be sent to all passive replicas for persistent storage. Since we are using Akka Cluster we can obtain a list of addresses for all replicas from the Cluster Extension, using the local replica ActorRef as a pattern into which remote addresses are inserted. The replicate function stores the update together with a required replication count into the replicating to, indexed by the update's sequence number. The update will stay in the queue until enough confirmations have been received in the form of Replicated messages as can be seen in the definition of the running behavior. Only when this count is reached is the update applied to the local storage and the confirmation sent back to the original client.

Update requests as well as confirmations may be lost on the way between network nodes. Therefore the active replica schedules a periodic reminder upon which it will resend all updates that are currently in the replicating queue. This ensures that eventually all updates are received by enough passive replicas. It is not necessary that all replicas receive the updates from the active one as you will see when looking at the implementation of a passive replica. The reason for this design choice is that burdening the active replica with all concerns of successful replication will not only make its implementation more complex but also increase the latency for responding to requests.

```
class Passive(askAroundCount: Int,
              askAroundInterval: FiniteDuration,
              maxLag: Int) extends Actor with ActorLogging {
  private val applied = Queue.empty[Replicate]

  // construct a name identifying this replica for data storage
  val selfAddress = Cluster(context.system).selfAddress
  val name = selfAddress.toString.replaceAll("[:/]", "_")
  val random = new Random

  def receive = readPersisted(name) match {
    case Database(s, kv) =>
      log.info("started at sequence {}", s)
      upToDate(kv, s + 1)
  }

  def upToDate(theStore: Map[String, JsValue],
               expectedSeq: Int): Receive = {
    case TakeOver(active) =>
      log.info("active replica starting at sequence {}", expectedSeq)
      active ! InitialState(theStore, expectedSeq)
```

```
    case Replicate(s, _, _, replyTo) if s - expectedSeq < 0 =>
      replyTo ! Replicated(s)
    case r: Replicate if r.seq == expectedSeq =>
      val nextStore = theStore + (r.key -> r.value)
      persist(name, expectedSeq, nextStore)
      r.replyTo ! Replicated(r.seq)
      applied.enqueue(r)
      context.become(upToDate(nextStore, expectedSeq + 1))
    case r: Replicate =>
      if (r.seq - expectedSeq > maxLag)
        fallBehind(expectedSeq, TreeMap(r.seq -> r))
      else
        missingSomeUpdates(theStore, expectedSeq, Set.empty,
                           TreeMap(r.seq -> r))
  }

  // implementation of fallBehind and missingSomeUpdates elided for now
}
```

The passive replica serves two purposes: it ensures the persistent storage of all incoming updates, and it maintains the current state of the full database so that the active replica can be initialized when required. When the passive replica starts up it first reads the persistent state of the database into memory including the sequence number of the latest applied update. As long as all updates are received in the right order only the third case of the up to date behavior will be invoked, applying the updates to the local store, persisting it, confirming successful replication, and changing behavior to expect the update with the following sequence number. Updates that are retransmitted by the active replica will have a sequence number that is less than the expected one and therefore only be confirmed as they have already been applied. A TakeOver request from a newly initializing active replica can in this state be answered immediately.

But what happens when messages are lost? Besides ordinary message loss this could also be due to a replica being restarted: between the last successful persistence before the restart and the initialization afterwards any number of additional updates may have been sent by the active replica that were never delivered to this instance since it was inactive. Such losses can only be detected upon receiving a subsequent update. The size of the gap in knowledge can be determined by comparing the expected sequence number with the one contained in the update and if it is too large—as determined by the maxLag parameter—we consider this replica as having fallen behind, otherwise it is merely missing some updates. The difference between these two lies in how we recover from the situation.

```
class Passive(askAroundCount: Int,
              askAroundInterval: FiniteDuration,
              maxLag: Int) extends Actor with ActorLogging {
  private val applied = Queue.empty[Replicate]
  private var takeOver = Option.empty[ActorRef]

  // initialization elided

  private var tickTask = Option.empty[Cancellable]
  def scheduleTick() = {
```

```
    tickTask foreach (_.cancel())
    tickTask = Some(context.system.scheduler.scheduleOnce(
      askAroundInterval, self, DoConsolidate)(context.dispatcher))
  }

  def caughtUp(theStore: Map[String, JsValue], expectedSeq: Int) {
    takeOver foreach (_ ! InitialState(theStore, expectedSeq))
    takeOver = None
    context.become(upToDate(theStore, expectedSeq))
  }

  def upToDate(theStore: Map[String, JsValue],
               expectedSeq: Int): Receive = {
    ... // cases shown previously elided
    case GetFull(replyTo) =>
      log.info("sending full info to {}", replyTo)
      replyTo ! InitialState(theStore, expectedSeq)
  }

  def fallBehind(expectedSeq: Int, _waiting: TreeMap[Int, Replicate]) {
    askAroundFullState()
    scheduleTick()
    var waiting = _waiting
    context.become {
      case Replicate(s, _, _, replyTo) if s < expectedSeq =>
        replyTo ! Replicated(s)
      case r: Replicate =>
        waiting += (r.seq -> r)
      case TakeOver(active) =>
        log.info("delaying active replica takeOver until upToDate")
        takeOver = Some(active)
      case InitialState(m, s) if s > expectedSeq =>
        log.info("received newer state at sequence {} (was at {})",
                 s, expectedSeq)
        persist(name, s, m)
        waiting.to(s).valuesIterator foreach (r =>
                                        r.replyTo ! Replicated(r.seq))
        val nextWaiting = waiting.from(expectedSeq)
        consolidate(m, s + 1, Set.empty, nextWaiting)
      case DoConsolidate =>
        askAroundFullState()
        scheduleTick()
    }
  }

  private def getMembers(n: Int): Seq[Address] = {
    val members = Cluster(context.system).state.members
    random.shuffle(members.map(_.address).toSeq).take(n)
  }
  private def askAroundFullState(): Unit = {
    log.info("asking for full data")
    for (addr <- getMembers(1)) replicaOn(addr) ! GetFull(self)
  }
  private def replicaOn(addr: Address): ActorSelection =
    context.actorSelection(self.path.toStringWithAddress(addr))
}
```

When falling behind we first ask a randomly selected replica for a full dump of the database and schedule a timer. Then we change behavior into a waiting state in which new updates are accumulated for later application, very old updates are immediately confirmed, and requests to take over are deferred. This state can only be left once an initial state message has been received; at this point we persist this newer state of the database, confirm all accumulated updates whose sequence number is smaller than the

now expected one, and try to apply all remaining updates by calling the consolidate function that is shown in the following:

```
  private val matches = (p: (Int, Int)) => p._1 == p._2

  private def consolidate(theStore: Map[String, JsValue],
                          expectedSeq: Int,
                          askedFor: Set[Int],
                          waiting: TreeMap[Int, Replicate]): Unit = {
    // calculate length of directly applicable queue prefix
    val prefix =
      waiting.keysIterator
        .zip(Iterator from expectedSeq)
        .takeWhile(matches)
        .size

    val nextStore =
      waiting.valuesIterator
        .take(prefix)
        .foldLeft(theStore) { (store, replicate) =>
          persist(name, replicate.seq, theStore)
          replicate.replyTo ! Replicated(replicate.seq)
          applied.enqueue(replicate)
          store + (replicate.key -> replicate.value)
        }
    val nextWaiting = waiting.drop(prefix)
    val nextExpectedSeq = expectedSeq + prefix

    // cap the size of the applied buffer
    applied.drop(Math.max(0, applied.size - maxLag))

    if (nextWaiting.nonEmpty) {
      // check if we fell behind by too much
      if (nextWaiting.lastKey - nextExpectedSeq > maxLag)
        fallBehind(nextExpectedSeq, nextWaiting)
      else
        missingSomeUpdates(nextStore, nextExpectedSeq, askedFor,
                           nextWaiting)
    } else caughtUp(nextStore, nextExpectedSeq)
  }
```

The waiting parameter contains the accumulated updates ordered by their sequence number, so we first find the length of the prefix of this collection that matches the sequence of the next expected sequence numbers, then we persist, confirm, and apply the corresponding updates and drop them from the waiting list. If the list is now empty—which means that all accumulated updates had consecutive sequence numbers—we conclude that we have caught up with the active replica and switch back into up to date mode. Otherwise we again determine whether the gap of knowledge that remains is too large or whether there is remaining holes can be filled in individually. The latter is done by the behavior that is shown last:

```
class Passive(askAroundCount: Int,
              askAroundInterval: FiniteDuration,
              maxLag: Int) extends Actor with ActorLogging {
  private val applied = Queue.empty[Replicate]
  private var takeOver = Option.empty[ActorRef]

  // initialization elided
```

```
  def upToDate(theStore: Map[String, JsValue],
               expectedSeq: Int): Receive = {
    ... // cases shown previously elided
    case GetSingle(s, replyTo) =>
      log.info("GetSingle from {}", replyTo)
      if (applied.nonEmpty &&
          applied.head.seq <= s && applied.last.seq >= s) {
        replyTo ! applied.find(_.seq == s).get
      } else if (s < expectedSeq) {
        replyTo ! InitialState(theStore, expectedSeq)
      }
  }

  def missingSomeUpdates(theStore: Map[String, JsValue],
                         expectedSeq: Int,
                         prevOutstanding: Set[Int],
                         waiting: TreeMap[Int, Replicate]): Unit = {
    val askFor =
      (expectedSeq to waiting.lastKey).iterator
        .filterNot(seq =>
          waiting.contains(seq) || prevOutstanding.contains(seq))
        .toList
    askFor foreach askAround
    if (prevOutstanding.isEmpty) scheduleTick()
    val outstanding = prevOutstanding ++ askFor
    context.become {
      case Replicate(s, _, _, replyTo) if s < expectedSeq =>
        replyTo ! Replicated(s)
      case r: Replicate =>
        consolidate(theStore, expectedSeq, outstanding - r.seq,
                    waiting + (r.seq -> r))
      case TakeOver(active) =>
        log.info("delaying active replica takeOver until upToDate")
        takeOver = Some(active)
      case GetSingle(s, replyTo) =>
        if (applied.nonEmpty &&
            applied.head.seq <= s && applied.last.seq >= s) {
          replyTo ! applied.find(_.seq == s).get
        } else if (s < expectedSeq) {
          replyTo ! InitialState(theStore, expectedSeq)
        }
      case GetFull(replyTo) =>
        log.info("sending full info to {}", replyTo)
        replyTo ! InitialState(theStore, expectedSeq)
      case DoConsolidate =>
        outstanding foreach askAround
        scheduleTick()
    }
  }

  // other helpers elided

  private def askAround(seq: Int): Unit = {
    log.info("asking around for sequence number {}", seq)
    for (addr <- getMembers(askAroundCount))
      replicaOn(addr) ! GetSingle(seq, self))
  }
}
```

Here we finally use the queue of applied updates that we previously maintained. When concluding that we are missing some updates we enter this state with the knowledge of the next expected consecutive sequence number and a collection of future updates that cannot yet be applied. We use this knowledge to first create a list of sequence numbers that we are missing—we will have to ask around among the other

replicas in order to obtain the corresponding updates. Again we will schedule a timer to ask around again in case some updates are not received; in order to avoid asking for the same update repeatedly we must maintain a list of outstanding sequence numbers that we already asked for. Asking around is done by sending a get single request to a configurable number of passive replicas. Within this state we install a behavior that will confirm known updates, defer initialization requests from an active replica, reply to requests for a full database dump, and answer requests for specific updates from other replicas that are in the same situation whenever possible. Whenever a replication request is received it could be either a new one from the active replica or one that we asked for. In any case we merge this update into the waiting list and use the consolidate function to process all applicable updates and possibly switch back to the up to date mode.

This concludes the implementation of both the active and passive replica. In order to use them we will need to start a passive replica on every single cluster node in addition to starting the Cluster Singleton manager. Client requests can be sent to the active replica by using the Cluster Singleton Proxy helper, an actor that keeps track of the current singleton location by listening to the cluster membership change events. The full source code including a runnable demo application can be found on github.

### 12.5.3 The Pattern Revisited

The implementation of this pattern consists of four parts:

- a cluster membership service that allows discovery and enumeration of all replica locations
- a cluster singleton mechanism that ensures that only one active replica is running at all times
- the active replica that accepts requests from clients, broadcasts updates to all passive replicas, and answers them after successful replication
- a number of passive replicas that persist state updates and help each other out to recover from message loss

For the first two we used the facilities provided by Akka Cluster in this example since the implementation of a full cluster solution is rather complex and not usually done from scratch. There are many other implementations that can be used for this purpose, the only important qualities are listed above. The implementation of both types of replica is more likely to be customized and tailored to a specific purpose. We demonstrated the pattern with a use case that exhibits the minimal set of operations representative of a wide range of applications: the Get request stands for operations that do not modify the replicated state and which can therefore be executed immediately on the active replica, while the Put request characterizes operations whose effects must be replicated such that they are retained across failures.

The performance of this replication scheme is very good as long as no failures occur

since the active replica does not need perform coordination tasks; all read requests can be served without needing further communication while write requests only need to be confirmed by a large enough subset of all replicas. This allows writes performance to be balanced with reliability in that a larger replication factor reduces the probability of data loss while a smaller replication factor reduces the impact of slow responses from some replicas—by requiring N responses we are satisfied by the N currently fastest replicas.

During failures we will see two different modes of performance degradation. If a network node hosting a passive replica fails then there will be increased network traffic after its restart in order to catch up with the latest state. If the network host running the active replica fails there will be a time period during which no active replica will be running: it takes a while for the cluster to determine that the node has failed and to disseminate the knowledge that a new Singleton needs to be instantiated. These coordination tasks need to be performed carefully in order to be reasonably certain that the old Singleton cannot interfere with future operations even if its node later becomes reachable again after a network partition. For typical cloud deployments this process takes of the order of seconds, it is limited by the fluctuations in network transmission latency and reliability.

While discussing failure modes we must also consider edge cases that can lead to incorrect behavior. Imagine that the active replica fails after sending out an update and the next elected active replica does not receive this message. In order to notice that some information was lost the new active replica would need to receive an update with a higher sequence number but with the presented algorithm that will never happen. Therefore when it accepts the first update after the failover the new active replica will unknowingly reuse a sequence number that some other replicas have seen for a different update. This can be avoided by requiring all known replicas to confirm the highest known sequence number after a failover, which of course adds to the downtime.

Another problem is to determine when to erase and when to retain the persistent storage after a restart. The safest option is to delete and repopulate the database in order to not introduce conflicting updates after a network partition that separated the active replica from the surviving part of the cluster. On the other hand this will lead to a significant increase in network usage that is unnecessary in most cases and it would be fatal in case the whole cluster was shut down and restarted. This problem could be solved by maintaining an epoch counter that is increased for every failover so that a replica can detect that it has outdated information after a restart—for this the active replica would include its epoch and its starting sequence number in the replication protocol messages.

Depending on the use case a trade-off must be made between reliable operation, performance, and implementation complexity. It should be noted that it is impossible to implement a solution that works perfectly for all imaginable failure scenarios.

### 12.5.4 Applicability

## 12.6 Multiple-Master Replication Patterns

*«Keep multiple copies of a service running in different locations, accept modifications everywhere and disseminate all modifications among them.»*

With active–passive replication the basic mode of operation is to have a relatively stable active replica that processes read and write requests without further coordination, keeping the nominal case as simple and efficient as possible while requiring special action during failover. This means that clients will have to send their requests to the currently active replica with resulting uncertainty in case of the failure: since the selection of the active replica is done until further notice instead of per request the client will not know whether the request has been disseminated in case a failure occurs before the transmission of the confirmation message.

Allowing requests to be accepted at all replicas means that the client itself can participate in the replication and thereby obtain more precise feedback on the execution of its requests. The co-location of the client and the replica does not necessarily mean that both are running in the process, placing them in the same failure domain makes their communication more reliable and their shared failure model simpler even if this just means running both on the same computer or even in the same computing center. The further distributed a system becomes the more prominent are the problems inherent to distribution, exacerbated by increased communication latency and reduced transmission reliability.

There are several strategies for accepting requests at multiple active replicas which differ mainly in how they handle requests that arrive during a network partition. In the following we will look at three classes of these:

- The most consistent results are achieved by establishing *consensus* on the application of each single update at the cost of not processing requests while dealing with failures.
- Availability can be increased by accepting potentially conflicting updates during a partition and *resolving the conflicts afterwards*, potentially discarding updates that were accepted at either side.
- Perfect availability without data losses can be achieved by restricting the data model such that concurrent updates are *conflict-free* by definition.

### 12.6.1 Consensus-Based Replication

Given a group of people we have a basic understanding what consensus means: within the group all members agree on a fact and acknowledge that this agreement is unanimous. From personal experience we know that reaching consensus is a process that can take quite a bit of time and coordination effort, in particular if the fact starts out as being contentious, i.e. initially there are multiple competing facts from which a single one must be decided to be true.

In computer science the term consensus[109] means roughly the same thing but of course the definition is more precise: given a cluster of *N* nodes and a set of proposals *P1–Pm* (the initial "facts" in the human analogy above), every non-failing node will eventually decide on a single proposal *Px* without the possibility to revoke that decision and all non-failing nodes will have decided on the same *Px*. This means that in our example of a key–value store one node proposes to update a key's value and after the consensus protocol is finished there will be a consistent cluster-wide decision about whether the update was performed—or in which order relative to other updates. During this process some cluster nodes may fail and if the number of failing nodes is below the failure tolerance threshold of the algorithm then consensus can be reached, otherwise consensus is impossible; this is equivalent to requiring a quorum for senate decisions in order to prevent an absent majority from reverting the decision in the next meeting[110].

Footnote 109 see for example https://en.wikipedia.org/wiki/Consensus_(computer_science) for an overview

Footnote 110 Incidentally a similar analogy is used in the original description of the PAXOS consensus algorithms by Leslie Lamport in "The Part-Time Parliament" (ACM Transactions on Computer Systems16 (2): 133–169, May 1998), available at http://research.microsoft.com/en-us/um/people/lamport/pubs/lamport-paxos.pdf .

A distributed key–value store can be built by using the consensus algorithm to agree on a replicated log: any incoming updates are put into numbered rows of a virtual ledger and since all nodes eventually agree on which update is at which row all nodes can apply the updates to their own local storage in the same order, resulting in the same state once everything is said and done. Another way to look at this is that every node runs its own copy of a replicated state machine and based on the consensus algorithm all individual state machines make the same transitions in the same order as long as not too many of them fail along the way.

#### APPLYING THE PATTERN

There are several consensus algorithms and even more implementations to choose from. In the following we use an existing example from the CKite project[111] in which a key–value store is written as simply as this:

Footnote 111 see https://github.com/pablosmedina/ckite for the implementation of the library and https://github.com/pablosmedina/kvstore/blob/4af8618995f79769808feffc56c123f7a5067722/src/main/scala/ckite/kvstore/K the full sample source code

```
class KVStore extends StateMachine {

  private var map = Map[String, String]()
  private var lastIndex: Long = 0

  def applyWrite = {
    case (index, Put(key: String, value: String)) => {
      map.put(key, value);
      lastIndex = index
      value
    }
  }

  def applyRead = {
    case Get(key) => map.get(key)
  }

  def getLastAppliedIndex: Long = lastIndex

  def restoreSnapshot(byteBuffer: ByteBuffer) =
    map =
      Serializer.deserialize[Map[String, String]](byteBuffer.array())

  def takeSnapshot(): ByteBuffer =
    ByteBuffer.wrap(Serializer.serialize(map))
}
```

This class describes only the handling of Get and Put requests once they have been agreed upon by the consensus algorithm, which is why this implementation is completely free from this concern. Applying a Put request means updating the Map that stores the key–value bindings and returning the written value while applying a Get request will only return the currently bound value (or None if there is none).

Since applying all writes since the beginning of time can be a time-consuming process there is also support for storing a snapshot of the current state, noting the last applied request's index in the log file. This is done by the takeSnapshot() function which is called by the CKite library at configurable intervals. Its inverse—the restoreSnapshot() function—turns the serialized snapshot back into a Map, which happens in case the KVStore is restarted after a failure or maintenance downtime.

In order to make use of the KVStore class we need to instantiate it as a replicated state machine. CKite uses the RAFT[112] consensus protocol as we can see:

Footnote 112 see http://raftconsensus.github.io/

```
object KVStoreBootstrap extends App {
  val ckite =
    CKiteBuilder()
      .stateMachine(new KVStore())
      .rpc(FinagleThriftRpc)
      .build
```

```
  ckite.start()
  HttpServer(ckite).start()
}
```

The HttpServer class starts an HttpService in which HTTP requests are mapped into requests to the key–value store, supporting consistent reads (that are applied via the distributed log), local reads that just return the currently applied updates at the local node (which may be missing updates that are currently in flight), and writes (as discussed). The API for this library is straight-forward in this regard:

```
val consistentRead    = ckite.read(Get(key))
val possiblyStaleRead = ckite.readLocal(Get(key))
val write             = ckite.write(Put(key, value))
```

#### THE PATTERN REVISITED

Writing your own consensus algorithm is almost never a good idea, there are so many pitfalls and edge cases to be considered that using one of the sound and proven ones is a very good default. Due to the nature of separating a replicated log from the state machine that processes the log entries it is easy to get started with existing solutions, as demonstrated by the minimal amount of code necessary for implementing the KVStore example: the only parts that need to be written are the generation of the requests to be replicated and the state machine that processes them at all replica locations.

The advantage of consensus-based replication is that it is guaranteed to result in all replicas agreeing on the sequence of events and thereby on the state of the replicated data. It is therefore straight-forward to reason about the the correctness of the distributed program in the sense that it will not get into an inconsistent state.

The price for this peace of mind is that in order to avoid mistakes the algorithm must be conservative, it cannot boldly make progress in the face of arbitrary failures like network partitions or node crashes. Requiring a majority of the nodes to agree on an update before progressing to the next one does not only take time, it can also fail altogether during network partitions like a three-way split where none of the parts represents a majority.

### *12.6.2 Replication with Conflict Detection and Resolution*

If we want to change our replication scheme such that it can continue to operate during a transient network partition then we will have to make some compromises: obviously it is impossible to reach consensus without communication so if all cluster nodes shall make progress in accepting and processing requests at all times there may be conflicting actions performed.

Consider the example of the Storage component within our batch service that stores the execution status of some computing job. When the job is submitted it will be recorded

as “new”, then it will become “scheduled”, “executing” and finally “finished” (ignoring failures and retries for the sake of simplicity here). But another possibility is that the client who submitted the job decides to cancel its execution because the computation is no longer needed, perhaps because parameters have changed and a new job has been submitted. That would attempt to change the job status to “cancelled”, with further possible consequences depending on the current job status—it might be taken out of the scheduled queue or it might need to be aborted if currently executing.

Assuming that we want to make the Storage component as highly available as we can we might let it accept job status updates even during a network partition that separates the storage cluster into two halves and renders it unable to successfully apply a consensus protocol until communication is restored. If the Client Interface sends the write of the “cancelled” status to one half while the execution service starts running the job and therefore sets the “executing” status on the other half, then the two parts of the Storage component have accepted conflicting information. When the network partition is repaired and communication is possible again the replication protocol will need to figure out that this has occurred and react to it.

### APPLYING THE PATTERN

The most prominent tool for detecting whether cluster nodes have performed conflicting updates is called a *version vector*[113]. With this each replica can keep track of who updated the job status since the last successful replication by incrementing a counter:

Footnote 113 In particular we do not need a vector clock, see https://haslab.wordpress.com/2011/07/08/version-vectors-are-not-vector-clocks/ for a discussion (in short: we only need to track whether updates were performed by a replica, not how many of them). See also http://arxiv.org/abs/1011.5808 for a description of “Dotted Version Vectors” by Nuno Preguiça, Carlos Baquero, Paulo Sérgio Almeida, Victor Fonte, Ricardo Gonçalves (Nov 2010).

- the status starts out as “scheduled” with an empty version vector on both nodes A and B
- when the Client Interface updates the status on node A, this one will register it as “cancelled” with a version vector of <A:1> (all nodes that are not mentioned are assumed to have version zero)
- when the Executor updates the status on node B, it will be registered as “executing” with the version vector <B:1>
- when A and B compare notes after the partition has been repaired the status will be both “cancelled” and “executing” with version vectors <A:1> and <B:1> and since neither fully includes the other a conflict will be detected

The conflict will then have to be resolved one way or another. A SQL database will decide based on a fixed or configurable policy, for example storing a timestamp together with each value and picking the latest update; in this case there is nothing to code since the conflict resolution happens within the database, the user code can be written just like in the non-replicated case.

Another possibility is implemented by the Riak database[114] which presents both values to any client who subsequently reads the key affected by the conflict, requiring the client to figure out how to proceed and bring the datastore back into a consistent state by issuing an explicit write request with the merged value; this is called *read repair*. An example of how this is done is part of the Riak documentation[115].

Footnote 114 see http://docs.basho.com/riak/latest/theory/concepts/Replication/

Footnote 115 see http://docs.basho.com/riak/latest/dev/using/conflict-resolution/

In the Batch Service example we could employ domain-specific knowledge within our implementation of the Storage component: after a partition is repaired all replicas exchange version information for all intermediately changed keys and when noticing this conflict it will be clear that the client wished to abort the now executing job—the repair procedure would automatically change the job status to "cancelled" (with a version vector of <A:1,B:1> to document that this includes both updates) and ask the Executor to terminate the execution of this job. One possibility for implementing this scheme would be to use a datastore like Riak and perform read repair at the application level together with the separately stored knowledge about which keys were written to during the partition.

### THE PATTERN REVISITED

We have introduced conflict detection and resolution at the level of a key–value store or database where the concern of state replication is encapsulated by an existing solution in the form of a relational database management system or other datastore. In this case the pattern consists of recording all actions as changes to the stored data such that the storage product can detect and handle conflicts that arise from accepting updates during network partitions or other times of partial system unavailability.

When using server-side conflict resolution (as is done by popular SQL database products) the application code itself is freed from this concern at the cost of potentially losing updates during the repair process—choosing the most recent update means discarding all others. Client-side conflict resolution allows tailored reactions that may benefit from domain-specific knowledge, but on the other hand it makes application code more complex to write since all read accesses to a datastore managed in this fashion must be able to deal with getting multiple equally valid answers to a single query.

### 12.6.3 Conflict-Free Replicated Data Types

In the previous section we achieved perfect availability of our Batch Service's Storage component—where perfect means "it works as long as one replica is reachable"—at the cost of either losing updates or having to care about manual conflict resolution. We can improve on this even further but unfortunately this comes at another cost: it is not possible to avoid conflicts while maintaining perfect availability without restricting the data types that can be expressed.

As an example consider a counter that is to be replicated. Conflict-freedom would be achieved by making sure that an increment registered at any replica would eventually be visible (i.e. effective) at all replicas independent of other concurrent increments. Clearly it is not enough to simply replicate the counter value: if an increment of 3 is accepted by node A while an increment of 5 is accepted at node B then the value after replication would either signal a conflict or miss one of the updates—as discussed in the previous section. Therefore we split the counter into individual per-node sub-counters, where each node only ever modifies its own sub-counter. Reading the counter then means summing up all the per-node sub-counters[116]. In this fashion both increments of 3 and 5 would be effective as the per-node values cannot see conflicting updates, and after the replication is complete the total sum will have been incremented by 8.

Footnote 116 An implementation in the context of Akka Distributed Data can be studied at https://github.com/akka/akka/blob/v2.4-M2/akka-distributed-data/src/main/scala/akka/cluster/ddata/GCounter.scala—this type of counter can only grow, hence its name "GCounter".

With this example it becomes clear that it is possible to create a data structure that fulfills our goal, but the necessary steps in the implementation of this replicated counter are tailored to this particular use-case and cannot be used in general—in particular we relied upon the fact that summing up all sub-counters correctly expresses the overall counter behavior. This is possible for a wide range of data types including sets and maps but it fails wherever global invariants cannot be translated to local ones; for a set it is easy to avoid duplicates because that can be done upon each insertion, but constructing a counter whose value must stay within a given range requires coordination again.

The data types that we are talking about here are called *conflict-free replicated data types*[117] (CRDT) and are currently being introduced in a number of distributed systems libraries and datastores. In order to define such a data type we need to formulate a rule on how to merge two of its values into a new resulting value. Instead of detecting and handling conflicts such a data type knows how to merge concurrent updates so that no conflict occurs.

Footnote 117 See M. Shapiro et al, “A comprehensive study of Convergent and Commutative Replicated Data Types” (2011), https://hal.inria.fr/inria-00555588 for an overview and C. Baquero, “Specification of convergent abstract data types for autonomous mobile computing” (1997), http://haslab.uminho.pt/cbm/publications/specification-convergent-abstract-data-types-autonomous-mobile-computing for early ground work.

The most important properties of the merge function are that it is symmetric and monotonic: it must not matter whether we merge value1 with value2 or value2 with value1, and having merged two values into a third one then future merges must not ever go back to a previous state (example: if v1 and v2 were merged to v2 then any merge of v2 with another value must not ever result in v1—you can picture this as that the values follow some order and merging only ever goes forward in this order, never backward).

## APPLYING THE PATTERN

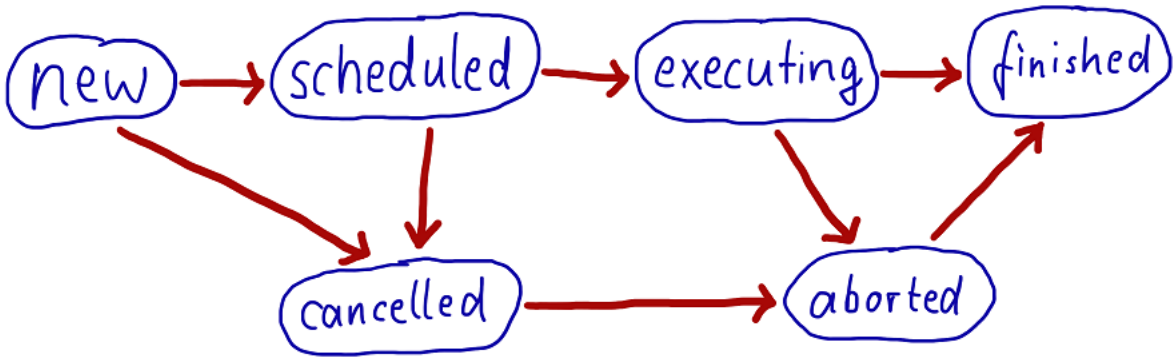


**Figure 12.11 Batch job status values as CRDT with their merge ordering indicated by state progression arrows: walking in the direction of the arrows goes from predecessor to successor in the merge order.**

Coming back to our example of updating the status of a batch job we will now demonstrate how a CRDT works. First, we define all possible status values and their merge order as shown in figure 12.11—such a graphical representation is the easiest way to get started when designing a CRDT with a small number of values. When merging two statuses there are three cases:

- if both statuses are the same then obviously we just pick that status
- if one of them is reachable from the other by walking in the direction of the arrows then we pick the one towards which the arrows are pointing; as an example merging “new” and “executing” will result in “executing”
- if that is not the case then we need to find a new status that is reachable from both by walking in the direction of the arrows, but we want to find the closest such status (otherwise “finished” would always be a solution, but not a very useful one); there is only one example in this graph, which is merging “executing” and “cancelled”, in which case we choose “aborted”—choosing “finished” would technically be possible and consistent but that choice would lose information (i.e. we want to retain both pieces of knowledge that are represented by “executing” and “cancelled”)

The next step is to cast this logic into code. Here we prepare the use of the resulting status representation with the Akka Distributed Data module that takes care of the replication and merging of CRDT values. The only thing needed is said merge function which is the only abstract method on the ReplicatedData interface:

```
final case class Status(val name: String)(_pred: => Set[Status],
                                          _succ: => Set[Status])
    extends ReplicatedData {

  type T = Status
  def merge(that: Status): Status = mergeStatus(this, that)

  lazy val predecessors = _pred
  lazy val successors = _succ
}

val New: Status =
            Status("new")(Set.empty, Set(Scheduled, Cancelled))
val Scheduled: Status =
            Status("scheduled")(Set(New), Set(Executing, Cancelled))
val Executing: Status =
            Status("executing")(Set(Scheduled), Set(Aborted, Finished))
val Finished: Status =
            Status("finished")(Set(Executing, Aborted), Set.empty)
val Cancelled: Status =
            Status("cancelled")(Set(New, Scheduled), Set(Aborted))
val Aborted: Status =
            Status("aborted")(Set(Cancelled, Executing), Set(Finished))
```

This is a trivial transcription of the graph from figure 12.11 where each node in the status graph is represented by Scala object with two sets of nodes describing the incoming and outgoing arrows, respectively: an arrow always goes from predecessor to successor, i.e. Scheduled is a successor of New and New is a predecessor of Scheduled. We could have chosen a more compact representation where each arrow is encoded only once, for example just providing the successor information and then after construction of all statuses a second pass would fill in the predecessor sets automatically; here we opted for being more explicit and saving the code for the post-processing step. Now it is time to look at the merge logic:

```
def mergeStatus(left: Status, right: Status): Status = {
  /*
   * Keep the left Status in hand and determine whether it is a
   * predecessor of the candidate, moving on to the candidate's
   * successor if not successful. The list of exclusions is used to
   * avoid performing already determined unsuccessful comparisons
   * again.
   */
  def innerLoop(candidate: Status, exclude: Set[Status]): Status =
    if (isSuccessor(candidate, left, exclude)) {
      candidate
    } else {
      val nextExclude = exclude + candidate
      val branches =
        candidate.successors.map(succ => innerLoop(succ, nextExclude))
      branches.reduce((l, r) =>
```

```
        if (isSuccessor(l, r, nextExclude)) r else l)
    }

  def isSuccessor(candidate: Status, fixed: Status,
                  exclude: Set[Status]): Boolean =
    if (candidate == fixed) true
    else {
      val toSearch = candidate.predecessors -- exclude
      toSearch.exists(pred => isSuccessor(pred, fixed, exclude))
    }

  innerLoop(right, Set.empty)
}
```

In this algorithm we merge two statuses, one called left and one called right. We keep the left value constant during the whole process and consider the right one as a candidate that we may need to move in the direction of the arrows. As an illustration consider merging New and Cancelled:

- If New is taken as the left argument then we will enter the inner loop with Cancelled being the candidate and the first conditional will call isSuccessor with the first two arguments being Cancelled and New; these are not equal, so the else branch will search all predecessors of Cancelled—which are New and Scheduled—whether one of them is a successor of New, which now satisfies the condition of candidate == fixed and therefore yields true, meaning that the candidate in innerLoop—Cancelled—will be returned as the merge result.
- If New is taken as the right argument then the first isSuccessor call will yield false and we enter the other branch in which both successors of our candidate New will be examined; trying Scheduled will be equally fruitless, escalating to Executing and Cancelled as its successors. Abbreviating the story a little we will eventually find that the merge result for the Executing candidate will be Aborted while for Cancelled it is Cancelled itself. These branches are then reduced into a single value by pairwise comparison and picking the predecessor, Cancelled in the case of trying Scheduled just now. Returning to the outermost loop invocation we thus get two times the same result of Cancelled for the two branches, which is also the end result.

This procedure is somewhat complicated by the fact that we allowed the two statuses of Executing and Cancelled to be unrelated to each other, necessitating the ability to find a common descendant. We will come back to why this is needed in our example, but first we take a look at how this CRDT is used by a hypothetical (and vastly oversimplified) Client Interface. In order to instantiate the CRDT we need to define a key that identifies it across the cluster:

```
// declare the Storage Component as mapping from String to Status
object StorageComponent extends Key[ORMap[Status]]("StorageComponent")
```

We need to associate a Status with each batch job and the most fitting predefined CRDT for this purpose is a so-called “observed-remove map”, in short ORMap. The name stems from the fact that only keys whose presence has previously been observed

can be removed from the map. Removal is a difficult operation since we have seen that CRDTs need a monotonic forward-moving merge function—just removing a key at one node and replicating the new map would only mean that the other nodes would add it right back during merges since that is the mechanism by which the key is spread across the cluster initially. For the details of how this is implemented see “An optimized conflict-free replicated set”[118].

Footnote 118 Annette Bieniusa, Marek Zawirski, Nuno Pregui¸ca, Marc Shapiro, Carlos Baquero, et al., https://hal.inria.fr/hal-00738680 (Oct 2012)

One thing to note here is that CRDTs can be composed as shown above: the ORMap uses Strings as keys (this is fixed by the Akka Distributed Data implementation) and some other CRDT as values. Instead of using our custom Status type we could have used an ORSet of PNCounters if we needed sets of counters to begin with, just to name one possibility. This makes it possible to create container data types with well-behaved replication semantics that are reusable in different contexts. Within the Client Interface—represented as a vastly oversimplified Actor below—we make use of the status map by referencing the StorageComponent key:

```
class ClientInterface extends Actor with ActorLogging {
  val replicator = DistributedData(context.system).replicator
  implicit val cluster = Cluster(context.system)

  def receive = {
    case Submit(job) =>
      log.info("submitting job {}", job)
      replicator !
        Replicator.Update(StorageComponent, ORMap.empty[Status],
                          Replicator.WriteMajority(5.seconds))
                         (map => map + (job -> New))
    case Cancel(job) =>
      log.info("cancelling job {}", job)
      replicator !
        Replicator.Update(StorageComponent, ORMap.empty[Status],
                          Replicator.WriteMajority(5.seconds))
                         (map => map + (job -> Cancelled))
    case r: Replicator.UpdateResponse[_] =>
      log.info("received update result: {}", r)
    case PrintStatus =>
      replicator ! Replicator.Get(StorageComponent,
                                  Replicator.ReadMajority(5.seconds))
    case g: Replicator.GetSuccess[_] =>
      log.info("overall status: {}", g.get(StorageComponent))
  }
}
```

The Replicator is the Actor that is provided by the Akka Distributed Data module that is responsible for running the replication protocol between cluster nodes. Most of the generic CRDTs like ORMap need to identify the originator of a given update and for that the Cluster extension is implicitly used—here it is needed by both function literals that modify the map during the handing of Submit and Cancel requests.

With the Update command we include not only the StorageComponent key, but also

the initial value should the CRDT not have been referenced before as well as a replication factor setting. This setting determines at which point the confirmation of a successful update will be sent back to the ClientInterface actor: we choose to wait until the majority of cluster nodes has been notified, but we could as well demand that all nodes have been updated or we could be satisfied once the local node has the new value and starts to disseminate it. The latter will be least reliable but also perfectly available (assuming that a local failure implies that the ClientInterface is affected as well), waiting for all nodes is most reliable but can easily fail; using the majority is a good compromise that works well in many situations—just as for legislative purposes.

The modifications performed by the Client Interface do not care about the previous job status, they just create a New entry or overwrite an existing one with Cancelled. The Executor component demonstrates some more interesting usage:

```
class Executor extends Actor with ActorLogging {
  val replicator = DistributedData(context.system).replicator
  implicit val cluster = Cluster(context.system)

  var lastState = Map.empty[String, Status]

  replicator ! Replicator.Subscribe(StorageComponent, self)

  def receive = {
    case Execute(job) =>
      log.info("executing job {}", job)
      replicator !
        Replicator.Update(StorageComponent, ORMap.empty[Status],
                          Replicator.WriteMajority(5.seconds),
                          Some(job))
                         { map =>
                           require(map.get(job) == Some(New))
                           map + (job -> Executing)
                         }
    case Finish(job) =>
      log.info("job {} finished", job)
      replicator !
        Replicator.Update(StorageComponent, ORMap.empty[Status],
                          Replicator.WriteMajority(5.seconds))
                         (map => map + (job -> Finished))
    case Replicator.UpdateSuccess(StorageComponent, Some(job)) =>
      log.info("starting job {}", job)
    case r: Replicator.UpdateResponse[_] =>
      log.info("received update result: {}", r)
    case ch: Replicator.Changed[_] =>
      val current = ch.get(StorageComponent).entries
      for {
        (job, status) <- current.iterator
        if (status == Aborted)
        if (lastState.get(job) != Some(Aborted))
      } log.info("aborting job {}", job)
      lastState = current
  }
}
```

When it is time to execute a batch job, the update request for the CRDT includes a request identifier (Some(job)) that has so far been left out: this value will be included in the success or failure message that the replicator will send back. The provided update

function now checks a precondition, namely that the currently known status of the given batch job is still New, otherwise the update will be aborted with an exception. Only upon receiving the UpdateSuccess message with this job name will the actual execution begin, otherwise a ModifyFailure will be logged (which is a subtype of UpdateResponse).

Finally, the Executor should actually abort batch jobs that were cancelled after being started. This is implemented by subscribing to change events from the replicator for the StorageComponent CRDT. Whenever there is a change, the replicator will take note of it and as soon as the (configurable) notification interval elapses a Replicator.Changed message will be sent with the current state of the CRDT. The Executor keeps track of the previously received state and can therefore determine which jobs have newly become Aborted. In this code sample we just log this, in a real implementation the Worker instance(s) for this job would be asked to terminate.

The full sample including the necessary cluster setup can be found in the source code archives for this chapter.

#### THE PATTERN REVISITED

Conflict-free replication allows perfect availability but requires the problem to be cast in terms of special data types, CRDTs. The first step is to determine which semantics are needed In our case we needed a tailor-made data type but there are a number of generically useful ones that are readily usable. Once the data type has been defined a replication mechanism must be used or developed that will disseminate all state changes and invoke the merge function wherever necessary. This could be a library as in the example shown in the previous section or it could be an off-the-shelf datastore based on CRDTs.

While it is easy to get started like this it should be noted that this solution cannot offer strong consistency: updates can occur truly concurrently across the whole system, making the value history of a given key non-linearizable. This may present a challenge in environments which are most familiar with and used to transactional behavior of a central authority—the centrality of this approach is precisely the limitation in terms of resilience and elasticity that conflict-free replication overcomes, at the cost of offering at most eventual consistency.

## *12.7 The Active–Active Replication Pattern*

*«Keep multiple copies of a service running in different locations and perform all modifications at all of them.»*

In the previous patterns we achieved resilience for the storage subsystem of our exemplary batch job processing facility by replicating it across different locations (data centers, availability zones, etc.). We saw that we can only achieve strong consistency when implementing a failover mechanism, both CRDT-based replication and conflict detection avoid this at the cost of not guaranteeing full consistency. One property of

failovers is that it takes some time: first we need to detect that there is trouble, then we need to establish consensus on how to fix it—e.g. switching to another replica—both these activities require communication and therefore cannot be completed instantaneously. Where this is not tolerable we need a different strategy, but since there is no magic bullet we shall expect to find different restrictions.

Instead of failing over as a consequence of detecting problems we will just assume that failures occur and therefore hedge our bets: instead of contacting only one replica we always perform the desired operation on all of them. If a replica does not respond correctly then we just conclude that it has failed and refrain from contacting it again—a new replica will be added based on monitoring and supervision.

The definition of active–active replication used here[119] is taken from the space industry, where for example measurements are performed using multiple sensors at all times and hardware-based voting mechanisms select the prevalent observation among them, discarding minority deviations by presuming them to be the result of failures. As an example the main bus voltage of a satellite might be monitored by a regulator that decides whether to drain the batteries or charge them with excess power coming from the solar panels; making the wrong decision in this regard will ultimately destroy the satellite and therefore three such regulators are taken together and their signals are fed into a majority voting circuit to obtain the final decision.

Footnote 119 It should be noted that database vendors sometimes use “active–active replication” to mean conflict detection and resolution as described in the previous section.

The drawback of this scheme is that we must assume that the inputs to all replicas are really the same so that the responses will also be the same, all replicas internally go through the same state changes “together”. In contrast to the concrete bus voltage that is measured in the satellite example—the one source of truth—having multiple clients contact three replicas of a stateful service means that there must be a central point that ensures that requests are delivered to all replicas in the same order. This will either be a single chokepoint (both concerning failures and throughput) or it will require costly coordination again. But instead of theorizing we shall look at a concrete example.

### 12.7.1 The Problem Setting

Again we apply this replication scheme to the key–value store that represents the Storage component of the Batch Job Service and we represent the two involved subcomponents—a coordinator and a replica—as vastly simplified Actors, concentrating on the basic working principle. The idea behind the the pattern is that all replicas go through the same state changes “in lockstep” without actually coordinating their actions and while running fully asynchronously. Since coordination is necessary nevertheless we need to control the requests that are sent to the replicas by introducing a middleman who also acts as bookkeeper and supervisor.

### 12.7.2 Applying The Pattern

The starting point for implementing this solution are the replicas themselves, which due to the lack of coordination can be kept quite simple:

```
private case class SeqCommand(seq: Int, cmd: Command,
                              replyTo: ActorRef)
private case class SeqResult(seq: Int, res: Result,
                             replica: ActorRef, replyTo: ActorRef)

private case class SendInitialData(toReplica: ActorRef)
private case class InitialData(map: Map[String, JsValue])

class Replica extends Actor with Stash {
  var map = Map.empty[String, JsValue]

  def receive = {
    case InitialData(m) =>
      map = m
      context.become(initialized)
      unstashAll()
    case _ => stash()
  }

  def initialized: Receive = {
    case SeqCommand(seq, cmd, replyTo) =>
      // tracking of sequence numbers and resends is elided here
      cmd match {
        case Put(key, value, r) =>
          map += key -> value
          replyTo ! SeqResult(seq, PutConfirmed(key, value), self, r)
        case Get(key, r) =>
          replyTo ! SeqResult(seq, GetResult(key, map get key), self, r)
      }
    case SendInitialData(toReplica) => toReplica ! InitialData(map)
  }
}
```

First we define sequenced command and result wrappers for the communication between the coordinator and the replicas as well as initialization messages to be sent between replicas. The replica starts out in a mode where it waits for a message containing the initial state to start out from—we must be able to bring new replicas online in the running system. Once the initialization data have been received the replica switches to its initialized behavior and replays all previously stashed commands. In addition to Put and Get requests it also understands a command to send the current contents of the key–value store to another replica in order to get that one initialized.

As noted within the code sample we have left out all sequence number tracking and resend logic (also from the coordinator Actor discussed below) in order to concentrate on the essence of this pattern. Since we have solved reliable delivery of updates for active–passive replication already we consider this part of the problem solved, please refer back to that section; in contrast to having the replicas exchange missing updates among each other we would in this case establish the resend protocol only between the coordinator and each replica individually.

Assuming that all replicas perform their duties if they are fed the same requests in the same order we now need to fulfill that condition: it is the responsibility of the coordinator to broadcast the commands, handle and aggregate the replies as well as manage possible failures and inconsistencies. In order to nicely formulate this we will need to create an appropriate data type that represents the coordinator's knowledge about the processing status of a single client request:

```
private sealed trait ReplyState {
  def deadline: Deadline
  def missing: Set[ActorRef]
  def add(res: SeqResult): ReplyState
  def isFinished: Boolean = missing.isEmpty
}

private case class Unknown(deadline: Deadline, replies: Set[SeqResult],
                           missing: Set[ActorRef], quorum: Int)
             extends ReplyState {
  override def add(res: SeqResult): ReplyState = {
    val nextReplies = replies + res
    val nextMissing = missing - res.replica
    if (nextReplies.size >= quorum) {
      val answer =
        replies.toSeq
          .groupBy(_.res)
          .collectFirst { case (k, s) if s.size >= quorum => s.head }
      if (answer.isDefined) {
        val right = answer.get
        val wrong =
          replies.collect {
            case SeqResult(_, res, replica, _) if res != right =>
              replica
          }
        Known(deadline, right, wrong, nextMissing)
      } else if (nextMissing.isEmpty) {
        Known.fromUnknown(deadline, nextReplies)
      } else Unknown(deadline, nextReplies, nextMissing, quorum)
    } else Unknown(deadline, nextReplies, nextMissing, quorum)
  }
}

private case class Known(deadline: Deadline, reply: SeqResult,
                         wrong: Set[ActorRef], missing: Set[ActorRef])
             extends ReplyState {
  override def add(res: SeqResult): ReplyState = {
    val nextWrong =
      if (res.res == reply.res) wrong else wrong + res.replica
    Known(deadline, reply, nextWrong, missing - res.replica)
  }
}
private object Known {
    def fromUnknown(deadline: Deadline,
                    replies: Set[SeqResult]): Known = {
      // did not reach consensus on this one, pick simple majority
      val counts = replies.groupBy(_.res)
      val biggest = counts.iterator.map(_._2.size).max
      val winners = counts.collectFirst {
        case (res, win) if win.size == biggest => win
      }.get
      val losers = (replies -- winners).map(_.replica)
      Known(deadline, winners.head, losers, Set.empty)
    }
  }
```

This ReplyState tracks when the time limit for a client response expires, whether the reply value is already known, which replica's response deviated from the prevalent one and which replica's response is still outstanding. When a new request is made we start out with Unknown reply state containing an empty set of replies and a set of missing replica ActorRefs that contains all current replicas. As responses from replicas are received our knowledge grows and this is represented by the add() function: the response is added to the set of replies and as soon as the required quorum of replicas has responded with a consistent answer the ReplyState switches to Known—taking note of the replica ActorRefs from which the wrong answer was received. If after receiving the last expected response still no answer has reached the quorum one of the answers must be selected in order to make progress, in this case we just use a simple majority as implemented in the fromUnknown function. Within the Known state we still keep track of arriving responses so that corrupted replicas can be detected. Before we dive into this aspect we take a look at the overall structure of the Coordinator:

```
class Coordinator(N: Int) extends Actor {
  private var replicas = (1 to N).map(_ => newReplica()).toSet
  private val seqNr = Iterator from 0
  private var replies = TreeMap.empty[Int, ReplyState]
  private var nextReply = 0

  override def supervisorStrategy = SupervisorStrategy.stoppingStrategy

  private def newReplica(): ActorRef =
    context.watch(context.actorOf(Replica.props))

  // schedule timeout messages for quiescent periods
  context.setReceiveTimeout(1.second)

  def receive = ({
    case cmd: Command =>
      val c = SeqCommand(seqNr.next, cmd, self)
      replicas foreach (_ ! c)
      replies += c.seq -> Unknown(5 seconds fromNow, Set.empty,
                                  replicas, (replicas.size + 1) / 2)
    case res: SeqResult if replies.contains(res.seq) &&
                           replicas.contains(res.replica) =>
      val prevState = replies(res.seq)
      val nextState = prevState.add(res)
      replies += res.seq -> nextState
    case Terminated(ref) =>
      replaceReplica(ref, terminate = false)
    case ReceiveTimeout =>
  }: Receive) andThen { _ =>
    doTimeouts()
    sendReplies()
    evictFinished()
  }

  // definition of doTimeouts, sendReplies and evictFinished withheld
}
```

In our simplified code sample the Coordinator directly creates the replicas as child actors; a real implementation would typically request the infrastructure to provision and

start replica nodes who would then register themselves with the Coordinator once the Replica is running on them. The Coordinator also registers for lifecycle monitoring of all replicas using context.watch() in order to be able to react to permanent failures that are detected by the infrastructure—in the case of Akka this service is implicitly provided by the Cluster module. Another thing to note is that in this example the Coordinator is the parent actor of the replicas and therefore also their supervisor; since failures escalated to the Coordinator usually imply that messages have been lost and our simplified example assumes reliable delivery we install a supervisor strategy that will terminate any failing child actor, eventually leading to the reception of a Terminated message upon which a new replica will replace the previously terminated one.

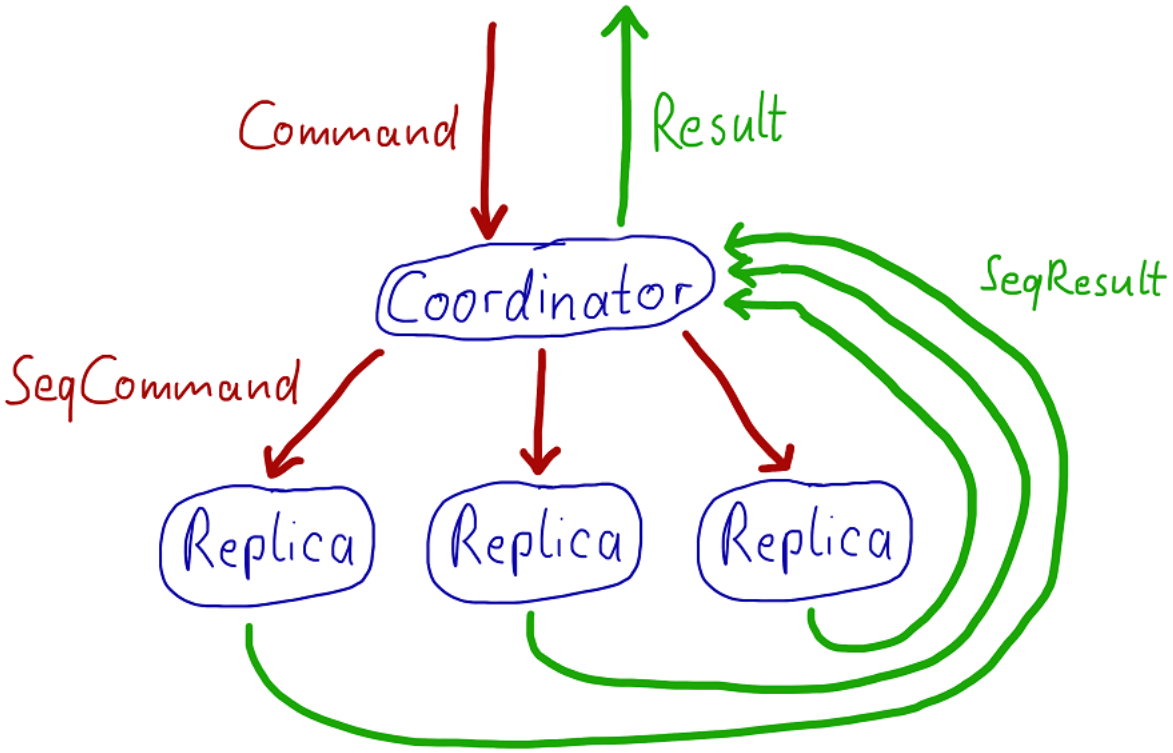


**Figure 12.12 Flow of messages in the active–active replication pattern.**

The flow of messages through the Coordinator is depicted in figure 12.12: while the external client sends Commands and expects back Results looping the requests through the replicas requires some additional information, hence the messages are wrapped as SeqCommand and SeqResult, respectively. The name signifies that these are properly sequenced, although as discussed we omit the implementation of reliable delivery that would normally be based on the contained sequence numbers. The only sequencing aspect that is modeled is that the external client shall see the results in the same order in which their corresponding commands were delivered; this is the reason for the nextReply variable that is used by the implementation of sendReplies():

```
@tailrec private def sendReplies(): Unit =
  replies.get(nextReply) match {
    case Some(k @ Known(_, reply, _, _)) =>
      reply.replyTo ! reply.res
      nextReply += 1
      sendReplies()
    case _ =>
  }
```

If the next reply to be sent has a known value, then send it back to the client and move on to the next one. This method is called after every handled message in order to flush replies to the clients whenever they are ready. The overall flow of replies through the Coordinator's response tracking queue (implemented by a TreeMap indexed by command sequence) is shown in figure 12.13.

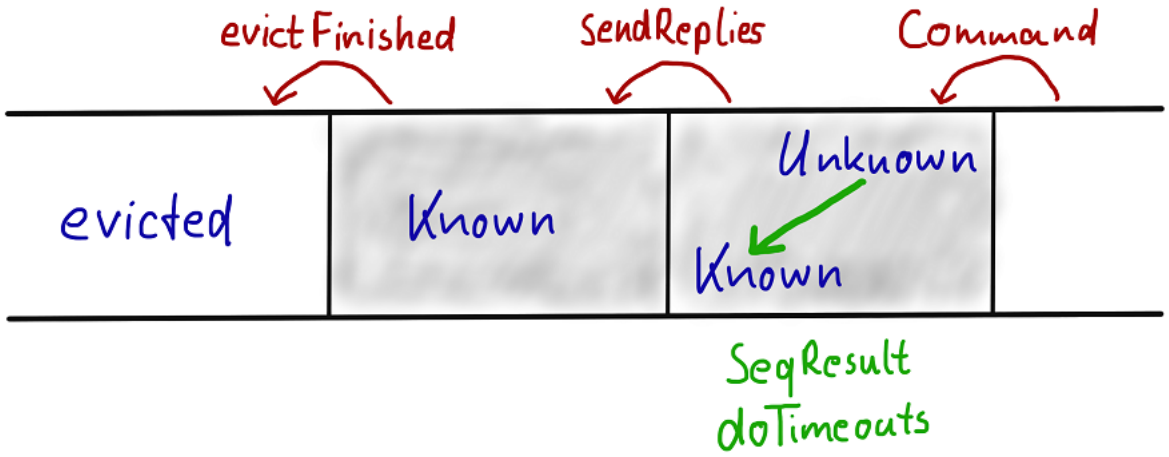


**Figure 12.13 The movement of replies through the status tracking within the Coordinator: new entries are generated whenever a Command is received, they move from Unknown to Known status by either receiving SeqResult messages or due to timeout, consecutive Known results are sent back to the external client, and replies where no more replica responses are expected are evicted from the queue.**

We have seen the handling of SeqResult messages already in the Coordinator's behavior definition, leaving the doTimeouts() function as the other possibility through which Unknown reply status can be transformed into a Known reply:

```
private def doTimeouts(): Unit = {
  val now = Deadline.now
  val expired = replies.iterator.takeWhile(_._2.deadline <= now)
  for ((seq, state) <- expired) {
    state match {
      case Unknown(deadline, received, _, _) =>
        val forced = Known.fromUnknown(deadline, received)
        replies += seq -> forced
      case Known(deadline, reply, wrong, missing) =>
        replies += seq -> Known(deadline, reply, wrong, Set.empty)
    }
  }
}
```

Since sequence numbers are allocated in strictly ascending order and all commands have the same timeout, replies will also time out in the same order. Therefore we can obtain all currently expired replies by computing the prefix of the replies queue for which the expiry deadline lies in the past. We turn each of these entries into a Known one for which no more responses are expected—even wrong replies that come in late are simply discarded, we will notice corrupted replicas during one of the subsequent requests. If no result has been determined yet for a command we again use the fromUnknown function to select a result with simple majority, again taking note of which replicas responded with a different answer (that is wrong by definition). The last remaining step is similar to a debriefing: for every command that we responded to we must check for deviating responses and replace their originating replicas immediately:

```
@tailrec private def evictFinished(): Unit =
  replies.headOption match {
    case Some((seq, k @ Known(_, _, wrong, _))) if k.isFinished =>
      wrong foreach (replaceReplica(_, terminate = true))
      replies -= seq
      evictFinished()
    case _ =>
  }

private def replaceReplica(r: ActorRef, terminate: Boolean): Unit =
  if (replicas contains r) {
    replicas -= r
    if (terminate) r ! PoisonPill
    val replica = newReplica()
    replicas.head ! SendInitialData(replica)
    replicas += replica
  }
```

The evictFinished function checks if the reply status of the oldest queued command is complete—i.e. there are no more responses expected—and if so it initiates the replacement of all faulty replicas and removes the status from the queue, repeating this process until the queue is empty or an unfinished reply status is encountered. Replacing a replica would normally mean asking the infrastructure to terminate the corresponding machine and provision a new one, but in this simplified example we just terminate the child actor and create a new one.

In order to get the new replica up to speed we need to provide it the current replicated state. One simple possibility is to ask one of the remaining replicas to transfer its current state to the new replica. Since this message will be delivered after all currently outstanding commands and before any subsequent commands, this state will be exactly the one that is needed by the new replica in order to be included in the replica set for new commands right away—the stashing and replay of these commands within the Replica actor has exactly the right semantics. In a real implementation there would need to be a timeout and resend mechanism for this initialization to cover cases where the replica that is supposed to transfer its state fails before it can complete the transmission.

It is important to note that the faulty replica is excluded from being used as the source of the initialization data, just like the new replica is.

### 12.7.3 The Pattern Revisited

This pattern was born in a conversation with some software architects working at a financial institution and it solves a rather specific problem: how can we keep a service running in fault-tolerant fashion with fully replicated state while not suffering from costly consensus overhead and avoiding any downtime during failure—not even allowing a handful of milliseconds for failure detection and failover?

The solution consists of two parts: the replicas execute commands and generate results without regards to each other, and the coordinator ensure that all replicas receive the same sequence of commands. Faulty replicas are detected by comparing the responses received to each individual command and flagging deviations. In the sample implementation given above this was the only criterion, a variation might be to also replace replicas that are consistently failing to meet their deadlines.

A side-effect of this pattern is that external responses can be generated as soon as there is agreement on what that response shall be, which means that requiring only 3 out of 5 replicas would allow shortening the usually long tail of the latency distribution. Assuming that slow responses are not correlated between the replicas (i.e. they are not caused by the specific properties of the command or otherwise related) then the probability of more than 2 replicas exceeding their 99th percentile latency is only 0.001%, which naïvely means that the individual 99th percentile is the cluster's 99.999th percentile[120].

---

Footnote 120 This is of course too optimistic since especially outliers in the latency distribution are usually caused by something that might well be correlated between machines, for example GC pauses for JVMs that were started at the same time and execute the same program with the same inputs will tend to occur roughly at the same time as well.

---

### 12.7.4 The Relation to Virtual Synchrony

This pattern is similar in some aspects to the *virtual synchrony*[121] work done by Ken Birman *et al* in the 1980's. The goal of both is to allow replicated distributed processes to run as if synchronized and to make the same state transitions in the same order. While we have restricted and simplified the solution by requiring a central entry point—our Coordinator—the virtual synchrony model postulates no such choke point. As we discussed under multiple-master replication this would normally require a consensus protocol to be used that ensures that transitions only occur once all replicas have acknowledged that they will do so. Unfortunately, this approach is doomed to fail as proven by Fischer, Lynch, and Paterson in what is usually referred to as the FLP[122] result. The practical probability for this kind of failure is vanishingly small but it is enough to question the endeavor of trying to provide perfect ordering guarantees in a distributed system.

Footnote 121 https://en.wikipedia.org/wiki/Virtual_synchrony, see https://www.cs.cornell.edu/ken/history.pdf for an introduction and historical discussion.

Footnote 122 Michael Fischer, Nancy Lynch, Michael Paterson: "Impossibility of Distributed Consensus with One Faulty Process", ACM, April 1985, http://cs-www.cs.yale.edu/homes/arvind/cs425/doc/fischer.pdf.

Virtual synchrony avoids this limitation by noticing that in most cases the ordering of processing two requests from different sources is not important: if the effects of two requests A and B are commutative then it does not matter whether they are applied in the order A, B or B, A because the state of the replica will be identical in the end. This is similar to how CRDTs achieve perfect availability without conflicts: impossibility laws do not matter if the available data types and operations are restricted in a way that conflicts cannot arise.

Taking a step back and considering our daily life we usually think in cause and effect: I observe that my wife's coffee mug is empty, fill it with coffee and tell her about it, expecting that when she looks at the mug she will be pleased because it is now full. By waiting with telling her until after my refill I made sure that the laws of physics—in particular causality—will ensure the desired outcome (barring catastrophes). This kind of thinking is so ingrained that we like to think about everything in this fashion, including distributed systems. When using a transactional database causality is ensured by the serializability of all transactions, they happen as if one were executed after the other, every transaction seeing the complete results of all previously executed transactions. While this works great as a programming abstraction it is in fact stronger than needed, causality does not imply that things happen in one universal order. In fact the laws of special relativity clearly describe which events can be causally related and which cannot—events that happen at remote locations are truly concurrent if one cannot fly from one to the other even at the speed of light.

This fact may serve to motivate the finding that causal consistency is the best that we can implement in a distributed system[123]. Virtual synchrony achieves greater resilience and less coordination overhead than consensus-based replication by allowing messages that are not causally related to be delivered to the replicas in random order. In this way it can reach similar performance than the active–active replication we describe above at the cost of carefully translating the desired program such that it relies only on causal ordering. If the program cannot be written in this fashion because for example some effects are not commutative, then at least for these operations the coordination cost of establishing consensus must be paid.

Footnote 123 Lloyd, Freedman, Kaminsky, Andersen: «Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS», SOSP'11, ACM 2011

In this sense the Coordinator represents a trade-off that introduces a single choke-point in exchange for being able to run arbitrary algorithms in a replicated fashion without the need for adaptation.

## 12.8 Summary

In this chapter we have covered a lot of ground on the design and implementation of resilient systems. We started out with describing simple components that obey the single responsibility principle, we observed the application of hierarchical failure handling in practice while implementing the error kernel pattern, we took note of the implications of relying on component restarts to recover from failure when contemplating the let-it-crash philosophy. Then we learnt how to decouple components from each other using circuit breakers for either side's protection and finally we discussed various replication patterns that allow us to distribute our systems in space so as to not put all eggs in one basket. Up to the circuit breaker the patterns are generically applicable but when it comes to replication patterns we are presented with a choice: do I favor consistency, reliability or availability? The answer depends on the requirements of the use-case at hand and it will rarely be black and white—there is a continuous range between these extremes and most patterns are even tunable. The following list can be used for orientation:

- *Active–passive replication* is relatively simple to use based upon an existing cluster singleton implementation; it is fast during normal operation, suffers downtime during failovers and offers good consistency due to having a single active replica.
- *Consensus-based replication* enables greater resilience by allowing updates to be accepted by any replica but in return for offering perfect consistency it suffers a rather high coordination overhead and consequently low throughput; preferring consistency entails being unavailable during severe failures.
- *Replication based on conflict detection and resolution* allows the system to stay available during severe failure conditions, but this can lead to data losses or require manual conflict resolution.
- *Conflict-free replicated data types* are formulated such that conflicts cannot arise by construction, therefore this scheme can achieve perfect availability even during severe

failures; the data types are restricted, requiring special adaptation of the program code as well as designing it for an eventual consistency model.

- *Active–active replication* addresses the concern of avoiding downtime during failures while maintaining a generic programming model; the cost is that all requests must be sent through a single choke-point in order to guarantee consistent behavior of all replicas—alternatively the program can be recast in terms of causal consistency in order to achieve high performance and high availability by employing virtual synchrony.

This summary is of course grossly simplified, please refer back to the individual sections for a more complete discussion of the limitations of each approach.

# 13 Resource Management Patterns

One concern that most of our systems share is that we need to manage or represent resources: file storage space, computation power, access to databases or web APIs, physical devices like printers or card readers, and many more. A component that we create might provide such a resource to the rest of the system on its own or we might need to incorporate external resources. In this chapter we discuss patterns for dealing with resources in reactive applications, in particular we present:

- the Resource Encapsulation Pattern
- the Resource Loan Pattern
- the Complex Command Pattern
- the Resource Pool Pattern
- Patterns for Managed Blocking

In the previous chapter we introduced the example of the Batch Job Service, a system that allows clients to submit computation jobs to have them executed by a fleet of elastically provisioned worker nodes. We focused on the hierarchical decomposition and the failure handling of such a system. Now we take a closer look at the provisioning and management of the worker nodes—these are the primary resource that is managed by the Batch Job Service.

## 13.1 Resource Encapsulation Pattern

*«A resource and its lifecycle is a responsibility that must be owned by one component.»*

From the Simple Component Pattern we know that every component does only one thing, but does it in full; in other words each component is fully responsible for the functionality that it provides to the rest of the system. If we regard that functionality as a resource that is used by other components—within or without the system—then it is clear that *resource*, *responsibility* and *component* exactly coincide, these three terms all denote the selfsame boundary. In other words: in this view resource encapsulation and

the single responsibility principle are the same. The same reasoning can be applied when considering other resources, in particular those that are used to provide a component's function. These are not implemented but merely managed or represented and the essence of the Resource Encapsulation Pattern is that we must identify that component into whose responsibility each resource falls and place it there—the resource becomes part of that component's responsibility. Sometimes this will lead you to identify the management of an external resource as a notable responsibility that needs to be broken out into its own Simple Component.

This pattern is closely related to the principles of hierarchical decomposition (chapter 6) and delimited consistency (chapter 8), you may wish to refresh your memory of these topics before diving in.

### 13.1.1 The Problem Setting

Recall the architecture of the Batch Job Service: the Client Interface offers the overall functionality to external clients and represents them internally, the Job Scheduling component decides which of the submitted jobs to execute in which order, the Execution component takes care of actually running the scheduled jobs, and beneath all these the Storage component allows the rest of the system to keep track of job status changes. Within the Execution component we identified two responsibilities, namely the interaction with the data center infrastructure and the individual worker nodes that are provisioned by that infrastructure. Each worker node is a resource that must be managed by the Execution component—we take over ownership and thereby responsibility by receiving these nodes from the infrastructure. The infrastructure itself is another resource that we merely represent within our system. We will explore how both of these are expressed in code in the next section.

### 13.1.2 Applying the Pattern

We apply this pattern by considering the process that the Execution component will use in order to manage the lifecycle of a worker node. After having introduced the main management processes we will see which pieces belong together and where they shall best be placed.

When the need arises to add a node to the computation cluster the infrastructure will need to be informed. There are many different ways to implement this, for example by utilizing a resource negotiation framework like MESOS, by directly interacting with a cloud provider like Amazon EC2 or Google Compute Engine, or using a custom mechanism accessible by some network protocol (e.g. an HTTP API). While all of these will need to send requests across the network they often present their client interface in the form of a library that can conveniently be used from your programming language of

choice. When the Execution component starts up it will need to initialize the interaction with the infrastructure provider, typically by reading access keys and network addresses from its deployment configuration.

An extremely simplified example of how a new worker node could be created is shown below using the Amazon Web Services API for EC2[124] with the Java language binding:

Footnote 124 see «Amazon Elastic Compute Cloud Documentation» at http://aws.amazon.com/documentation/ec2/

```
public Instance startInstance(AWSCredentials credentials) {
  AmazonEC2Client amazonEC2Client = new AmazonEC2Client(credentials);

  // ask for the creation of exactly one new node
  RunInstancesRequest runInstancesRequest =
    new RunInstancesRequest()
      .withImageId("my-image-id-for-a-worker")
      .withInstanceType("m1.small")
      .withMinCount(1)
      .withMaxCount(1);

  RunInstancesResult runInstancesResult =
    amazonEC2Client.runInstances(runInstancesRequest);
  Reservation reservation = runInstancesResult.getReservation();
  List<Instance> instances = reservation.getInstances();

  // there will be exactly one instance in this list, otherwise
  // runInstances() would have thrown an exception
  return instances.get(0);
}
```

Having the instance descriptor we can obtain the private network address of this new worker node and start interacting with it; how that interaction looks like depends on the inter-component communication fabric we are using which could be as simple as an HTTP API[125]. Before we go there we need to consider the possibility that the Amazon Web Services may become unreachable or fail for some reason. The client library signals this by throwing an AmazonClientException that we will need to handle, possibly retrying the operation, switching into a degraded mode or escalating the failure. As discussed in section 12.4 we should also monitor the reliability of the cloud infrastructure using a circuit breaker to avoid making a large number of pointless requests within a short time period. All of this gets easier by lifting the functionality into a Future so that we can describe these aspects in an event-driven fashion:

Footnote 125 We expect the development of higher-level Service definition frameworks in the near future that will abstract over the precise communication mechanism and offer a consistent code representation of service interaction in a fully location transparent fashion.

```
// using Scala’s Future and Akka’s CircuitBreaker
import scala.PartialFunction;
import scala.concurrent.ExecutionContext;
import scala.concurrent.Future;
```

```
import akka.dispatch.Futures;
import akka.japi.pf.PFBuilder;
import akka.pattern.CircuitBreaker;

// we need a thread pool and a circuit breaker instance
private ExecutionContext exeCtx;
private CircuitBreaker circuitBreaker;

public Future<Instance> startInstanceAsync(AWSCredentials credentials) {
  Future<Instance> f = circuitBreaker.callWithCircuitBreaker(() ->
    Futures.future(() -> startInstance(credentials), exeCtx));

  // define recovery strategy: some AWS calls can safely be retried;
  // the circuit breaker will take care of recurring failures;
  // any unmatched exception will not be recovered
  PartialFunction<Throwable, Future<Instance>> recovery =
    new PFBuilder<Throwable, Future<Instance>>()
      .match(AmazonClientException.class,
             ex -> ex.isRetryable(),
             ex -> startInstanceAsync(credentials))
      .build();

  // decorate Future with recovery
  return f.recoverWith(recovery, exeCtx);
}
```

In this fashion we have wrapped up the task of instantiating a new worker node such that all failures are registered—tripping the circuit breaker when necessary—and those failures that are expected to routinely be fixed by trying again lead to retries. The assumption here is that such failures are complete (i.e. no partial success has already changed the system state) and transient. Refinements of this scheme might implement a backoff strategy that schedules retries for progressively delayed points in time instead of trying again immediately, it is easy to see that this would be incorporated by just using a scheduler call (e.g. using akka.pattern.after[126]) wrapping startInstanceAsync() in the recovery strategy—of course we do not block a thread from the ExecutionContext's thread pool by using Thread.sleep().

Footnote 126 see http://doc.akka.io/japi/akka/current/akka/pattern/Patterns.html for the Java documentation

The attentive reader will have noticed that in the code snippets we are using the synchronous version of the AmazonEC2Client even though there is an asynchronous version as well: AmazonEC2AsyncClient provides a runInstancesAsync() method that accepts a completion callback as its second parameter (the returned java.util.concurrent.Future is not useful for event-driven programming as discussed in chapter 3). We could use the callback to supply the value for a Promise and thereby obtain a Scala Future in an event-driven fashion:

```
public Future<RunInstancesResult> runInstancesAsync(
                                       RunInstancesRequest request,
                                       AmazonEC2Async client) {
  Promise<RunInstancesResult> promise = Futures.promise();
  client.runInstancesAsync(request,
    new AsyncHandler<RunInstancesRequest, RunInstancesResult>() {
      @Override
```

```
      public void onSuccess(RunInstancesRequest request,
                            RunInstancesResult result) {
        promise.success(result);
      }

      @Override
      public void onError(Exception exception) {
        promise.failure(exception);
      }
    });
  return promise.future();
}
```

Unfortunately the AWS library implements the asynchronous version in terms of the same blocking HTTP network library that also powers the synchronous version (based on the Apache HTTP client library), it just runs the code on a separate thread pool. We could configure that thread pool to be the same ExecutionContext that we use to run our Scala Futures by supplying it as a constructor argument when instantiating the AmazonEC2AsyncClient. That would not be a net win, however, since instead of just wrapping the synchronous call in a Future we would have to bridge all client methods in the fashion shown above—an overhead of 15–20 lines per API method; the execution mechanics would be the same while adapting the different asynchronous API styles would be significant extra programming effort (and hence also more opportunity for errors). We will take a deeper look at situations like this one later in this chapter when we discuss patterns for Managed Blocking.

Now that we have started the worker node we need to also manage the rest of its lifecycle: the Execution component needs to keep track of which workers are available, monitor their health by performing regular status checks, and finally shut them down when they are no longer needed. Performing health checks typically means making a service calls that queries some performance indicators the service is monitoring internally: the fact that a response is received at all signals general availability and the measured quantities can be factored into future decisions of whether to scale the number of worker nodes up or down—or it can also be indicative of specific problems like unusually high memory consumption that need dedicated reaction (for example an operator alert or automatic reboot after taking a diagnostic memory dump).

This brings us to the final step of a worker node's lifecycle, the Execution component needs to instruct the infrastructure to shut a node down. Completing the Amazon Web Services example this would be done like this:

```
public Future<TerminateInstancesResult> terminateInstancesAsync(
                        AmazonEC2Client client, Instance... instances) {
  List<String> ids = Arrays.stream(instances)
                           .map(i -> i.getInstanceId())
                           .collect(Collectors.toList());
  TerminateInstancesRequest request = new TerminateInstancesRequest(ids);

  Future<TerminateInstancesResult> f =
    circuitBreaker.callWithCircuitBreaker(
```

```
      () -> Futures.future(() -> client.terminateInstances(request),
                           executionContext)
    );

  PartialFunction<Throwable, Future<TerminateInstancesResult>> recovery =
    new PFBuilder<Throwable, Future<TerminateInstancesResult>>()
      .match(AmazonClientException.class,
             ex -> ex.isRetryable(),
             ex -> terminateInstancesAsync(client, instances))
      .build();
  return f.recoverWith(recovery, executionContext);
}
```

Of course we will want to use the same circuit breaker and ExecutionContext as for the runInstancesAsync() implementation above because it is the same infrastructure service that we are addressing—it is not reasonable to assume that creating and terminating machine instances are independent operations such that one keeps working while the other is systematically unavailable (as in failing to respond, not denying invalid input). Therefore we place the responsibility to communicate with the infrastructure service within its own Execution sub-component that we called Resource Pool Interface in section 12.3. While the AmazonEC2Client offers a rich and detailed API (we glossed over the creation of security groups, configuration of availability zones and key pairs etc.) the resource pool need only offer high-level operations like creating and terminating worker nodes. We represent only a tailored view onto the externally provided capabilities to the components within our own system and we do so via one single component dedicated to this purpose.

This has another important benefit in that we have not only encapsulated the responsibility for dealing with the vicissitudes of the external service's availability, we can also switch to a completely different infrastructure service provider by replacing this one internal representation. The Execution component does not need to know whether the worker nodes are running on Amazon's Elastic Compute Cloud or Google's Compute Engine (or whatever computing infrastructure is en vogue at the time you are reading this) as long as it can communicate with the services that the worker nodes provide.

Another aspect of this placement of responsibility is that this is the natural—and only—place where we could implement service call quota management: if the infrastructure API imposed some limits of how frequently we may make requests then we would keep track of the requests that pass through this one access path. This would allow us to delay requests in order to avoid a temporary excess that could lead to punitively degraded service—to our knowledge this is not true for Amazon Web Services but for other web APIs such limitations and enforcement are common. Instead of running into the quota on the external service we would degrade the internally represented service such that the external service is not burdened with too many requests.

To recapitulate, we have considered the management actions that the Execution component needs to perform in order to provision and retire worker nodes and we have

placed the responsibility of representing the infrastructure provider that performs these functions within a dedicated sub-component called the Resource Pool Interface. While the mechanism for conveying the requests and responses between the Execution component and its worker nodes will change depending on which services frameworks are available over time, the remaining aspect that we need to discuss in the context of the Resource Encapsulation Pattern is how to model the knowledge about and management of the worker nodes within the Execution component.

Each worker node will gather its own performance metrics and react to the failures that it can address, but ultimately it is the Execution component that is responsible for the currently running workers, it has taken this responsibility by asking the resource pool to provision them. There are classes of failures—like fatal resource exhaustion in terms of CPU cycles or memory—that cannot be dealt with from within, the supervising component needs to keep track of its subordinates and dispose of those that have failed terminally or are otherwise inaccessible. Another way to look at this is that a worker node does not only provide its own service to the rest of the system, it is also coupled to a resource that must be managed in addition to the service that the resource powers. This is true in all cases where the supervising component assumes this kind of responsibility by effecting the creation or by asking for the transfer of ownership of such a resource.

As a demonstration of the management of a worker node's underlying resource we sketch an actor that takes this responsibility:

```
class WorkerNode extends AbstractActor {
  private final Cancellable checkTimer;

  public WorkerNode(InetAddress address, FiniteDuration checkInterval) {
    checkTimer = getContext().system().scheduler()
                  .schedule(checkInterval, checkInterval,
                            self(), DoHealthCheck.instance,
                            getContext().dispatcher(), self());

    List<WorkerNodeMessage> msgs = new ArrayList<>();
    receive(ReceiveBuilder
            .match(WorkerNodeMessage.class, msgs::add)
            .match(DoHealthCheck.class, dhc -> { /* perform check */ })
            .match(Shutdown.class, s -> {
                msgs.stream().forEach(msg -> {
                  WorkerCommandFailed failMsg =
                    new WorkerCommandFailed("shutting down", msg.id());
                  msg.replyTo().tell(failMsg, self()));
                  /* ask Resource Pool to shut down this instance */
              })
            .match(WorkerNodeReady.class, wnr -> {
                /* send msgs to the worker */
                getContext().become(initialized());
              })
            .build());
  }

  private PartialFunction<Object, BoxedUnit> initialized() {
    /* forward commands and deal with responses from worker node */
  }

  @Override
```

```
  public void postStop() {
    checkTimer.cancel();
  }
}
```

In the spirit of delimited consistency as discussed in chapter 8 we bundle all aspects of interaction with the worker node in this representation so that the forwarding of messages to and from the worker node can take into account the worker's lifecycle changes and current health status. With this encapsulation the Execution component will just create a WorkerNode actor for every worker node that it asks the Resource Pool to create and then it only needs to communicate with that actor as it if was the worker node itself. This proxy hides the periodic health check processing as well as the fact that after the instance has been created it will take a certain amount of time for the worker's services to start up and signal their readiness for accepting commands.

When implementing the WorkerNode class we encountered the need to ask the Resource Pool to shut down the represented instance. In a full-fledged implementation we might want to add more features that need to interact with the Resource Pool, for example monitoring the instances also via the cloud infrastructure provider's facilities (in the example above that would be Amazon CloudWatch). This is another reason why placing the responsibility for all such interaction within a dedicated sub-component, otherwise we would start duplicating this code in several places and thereby lose the ability to monitor the availability of the cloud infrastructure service consistently in only one location. It should be noted that this is meant in a logical sense and not necessarily in a physical one: the Resource Pool Interface could well be replicated for fault tolerance, and in this case we would not even care about synchronizing the state that it maintains since losing the circuit breaker's status in the course of a component crash would not have a large or lasting negative effect.

### 13.1.3 The Pattern Revisited

We have examined the interactions between the Execution component and the infrastructure service that provisions the worker nodes and have placed all aspects of this interaction within a dedicated component, the Resource Pool Interface. It is this component's responsibility to represent the resource pool to the rest of the system, allowing consistent treatment of the infrastructure provider's availability and limitations. This encapsulation is also in accordance with the principle of abstracting over the concrete implementation of a potentially exchangeable resource, in this case we simplify the adaptation to a different cloud infrastructure provider.

The second aspect we illuminated was that the worker nodes are based upon dynamically provisioned resources that need to be owned by their supervising component. Therefore we placed the responsibility of monitoring the worker node and communicating with it in a WorkerNode sub-component of the Execution component,

sketched as an actor for illustration. While communication with the services provided by a worker node is taken care of by the service fabric or framework there is a remaining responsibility that cannot be satisfied from within the worker node since it is about the management of its underlying resources.

These are the two cases in which the Resource Encapsulation Pattern is used: to represent an external resource or to manage a supervised resource—both in terms of its lifecycle and its function, in accordance with the Simple Component Pattern and the principle of delimited consistency.

One aspect that we glossed over was the precise relation of the WorkerNode sub-components to their Execution parent: should a WorkerNode supervisor be its own component or should it just be bundled with the Execution component itself? Both approaches are certainly possible, the code modularity offered by object-oriented programming can express the necessary encapsulation of concerns just as well as deploying a WorkerNode service instance on the hardware resources that the Execution component is using—spinning up a separate node would again require us to establish a supervision scheme and therefore not solve the problem[127]. It will depend on the case at hand which way the decision is made, where influencing factors are

Footnote 127 It should be noted that this depends on the service framework used, though, in that automatic resource cleanup in combination with health monitoring might already be provided—meaning that this pattern is incorporated at the framework level.

- the complexity of the resource management task itself
- the runtime overhead of service separation for the chosen service framework
- the development overhead of introducing another asynchronous messaging boundary

In many cases it will be preferable to run the management sub-components within their parent's context (i.e. just encapsulate this aspect in a separate class or function library). When using an Actor-based framework it will typically be a good middle ground to separate the resource management out into its own Actor, making it look and behave like a separate component while sharing most of the runtime context and avoiding large runtime overhead.

### 13.1.4 Applicability

This is an architectural pattern that mainly informs the design of the component hierarchy and the placement of implementation details in code modules—either reinforcing the previously established hierarchical decomposition or leading to its refinement. The concrete expression in code depends on the nature of the resource that is being managed or represented. The pattern is applicable wherever resources are integrated into a system, in particular when these resources have a lifecycle that needs to be managed or represented.

In some cases the nature of resources that are used by the system is not immediately visible: in the example a beginner's mistake might be to leave the worker node instances to their own devices after being created, having them shut themselves down when no longer needed. This works well most of the time, but failure cases with lingering but defunct instances will manifest in surprisingly large infrastructure costs, at which point it becomes obvious that reliable lifecycle management is required.

## 13.2 Resource Loan Pattern

*«Give a client exclusive transient access to a scarce resource without transferring ownership.»*

A variant of the Resource Loan Pattern is widely used in non-reactive systems, the most prominent example being that of a database connection pool. Database access is represented by a connection object via which arbitrary operations can be performed. The creation of connections is expensive and their number is limited, therefore a connection is not owned by client code but taken from a pool before performing an operation and put back afterwards. The connection pool is responsible for managing the lifecycle of its connections and client code just obtains temporary permission to make use of them. Failures will in this scenario be communicated to the client but their effect on the connection in question is handled by the pool—the pool owns and supervises the connections.

In a reactive system we strive to minimize contention as well as the need for coordination, hence the classical database connection pool will usually only feature as an internal implementation detail of a component whose data storage is realized by means of a relational database. That notwithstanding we encounter the usage of scarce resources within our systems ubiquitously and the same philosophy that drives the connection pool abstraction is useful in reactive system design as well.

### 13.2.1 The Problem Setting

Towards the end of the discussion of the previous pattern we touched upon the possibility of separating the ownership and the usage of a resource: not being responsible for the supervision aspects frees the user from having to perform monitoring tasks or recovery actions. In the example of the Execution component of our batch job service it may seem extraneous that the WorkerNode sub-component needs to watch over the physical instance that was provisioned via the Resource Pool Interface. Would it not be nicer if the Resource Pool were not merely a messaging façade for talking to a cloud provider but also took responsibility for the lifecycle management of the instances it provisions? This is what we will investigate in the following.

### 13.2.2 Applying the Pattern

Before we look at this part in more detail we need to establish the terminology. The word "loan" is often understood in a financial context where a *lender* gives a certain sum to a *borrower* who is expected to pay it back at a later time, usually with interest. More generally this applies to any asset that can be transferred, with the important notions that ownership of the asset remains with the lender throughout the process and that the transfer is only temporary and will eventually be reversed. Renting a flat falls in this category, the landlord lets you live in their property and expects you to vacate it when the lease ends; meanwhile the landlord stays responsible for the general upkeep of the building and everything that was rented out together with it. This example also illustrates the exclusivity of the arrangement given that a flat can only be rented out to one tenant at a given time. Therefore the resource (the flat in this case) is also scarce: it cannot simply be copied or inhabited by multiple tenants independently at the same time, this resource comes at a per-instance cost.

To answer the question above we consider the worker nodes that are provisioned by the Resource Pool Interface to be like flats that the Execution component wants to use. It will place a worker inside each flat who will then process batch jobs, a worker node is a potential home for a batch job process. In this image flats themselves are provided by the cloud infrastructure, but this is just the most basic empty apartment you can think of, there is nothing interesting in there until someone moves in. The worker node components in our decomposition of the batch job service correspond to people who need a flat to live in, once a worker has moved into a place the Execution component can send work items to their address and receive replies from them—the business level information can flow. The missing piece is a kind of concierge who looks after the flats rented for workers and checks on them regularly to see that everything is in order, both with the property and with the worker as well. The latter allows the Execution component (the work provider) to concentrate entirely on conversations about the work that is to be done, monitoring of the workforce is done by the concierge. The concierge is responsible for ending the lease when a worker moves out, solving the problem of possibly leaking resources.

Switching back from the anthropomorphic metaphor into computer programming this means that when the Execution component asks the Resource Pool Interface for a new worker node the resource pool will use the cloud infrastructure to provision a new machine instance, for example using the Amazon Web Services EC2 API as shown in the previous pattern. But instead of just returning the instance identifier and network address to the Execution component the resource pool will now assume responsibility for the worker node, it will need to start monitoring the service by performing regular health checks. The Execution component will only receive the network address at which the

new worker node service is being provided and it will assume that the resource pool keeps this node in good working condition—or terminates it and provides a new one.

In order to do that the resource pool must know about the kind of service that is being provided by the worker node, it must be able to ask for the relevant set of performance metrics and it must understand their meaning to assess a worker node's fitness. The Resource Pool Interface thus assumes more responsibility as before and is therefore also more tightly coupled to the function of the resources it provides. Instead of a more or less generic representation of the cloud infrastructure API it becomes more specific, tailored to the needs of the batch job service; in return we achieve a better separation of concerns between the lender and the borrower. The relationship in the previous pattern was that of a manufacturer and a buyer, where the latter's obligation to perform maintenance led to a coupling that we can in this case avoid. In source code this means we will have WorkerNode representations within the Execution component as well as within the Resource Pool Interface component, but these take care of the different aspects that were previously mixed within one class:

```
// This is the representation used within the Execution component
class WorkerNodeForExecution extends AbstractActor {

  public WorkerNodeForExecution(InetAddress address) {
    List<WorkerNodeMessage> msgs = new ArrayList<>();
    receive(ReceiveBuilder
            .match(WorkerNodeMessage.class, msgs::add)
            .match(Shutdown.class, s -> {
                msgs.stream().forEach(msg -> {
                  WorkerCommandFailed failMsg =
                    new WorkerCommandFailed("shutting down", msg.id());
                  msg.replyTo().tell(failMsg, self()));
                  /* ask Resource Pool to shut down this instance */
              })
            .match(WorkerNodeReady.class, wnr -> {
                /* send msgs to the worker */
                getContext().become(initialized());
              })
            .build());
  }

  private PartialFunction<Object, BoxedUnit> initialized() {
    /* forward commands and deal with responses from worker node */
  }
}

// This is the representation used within the Resource Pool Interface
class WorkerNodeForResourcePool extends AbstractActor {
  private final Cancellable checkTimer;

  public WorkerNodeForResourcePool(InetAddress address,
                                   FiniteDuration checkInterval) {
    checkTimer = getContext().system().scheduler()
                  .schedule(checkInterval, checkInterval,
                            self(), DoHealthCheck.instance,
                            getContext().dispatcher(), self());
    receive(ReceiveBuilder
            .match(DoHealthCheck.class, dhc -> { /* perform check */ })
            .match(Shutdown.class, s -> { /* clean up this resource */ })
            .build());
  }
```

```
  @Override
  public void postStop() {
    checkTimer.cancel();
  }
}
```

### 13.2.3 The Pattern Revisited

While applying this pattern we have segregated the responsibilities of resource maintenance and usage: the Execution component asks for the service of a new worker node and only gets that back in response without the additional burden that comes with a transfer of ownership. The resource that is loaned in this fashion is still exclusively available to the borrower, the Execution component can keep usage statistics knowing that only those jobs will be processed by the worker node that it has sent there itself, there is no competition for this resource between different borrowers for the duration of the loan.

The price for this simplification of the borrower was that the lender now needs to take over the responsibilities that the borrower has shed, requiring the lender to know more about the resource that it is loaning. One important point is that this additional knowledge should be kept minimal lest we violate the Simple Component Pattern and entangle the functionalities of lender and borrower more than necessary. This is particularly relevant where different kinds of borrowers enter the picture, the purpose of separating the lender, borrower, and loaned resource is to keep their responsibilities segregated and as loosely coupled as is practical. The lender should not know more about the capabilities of the resource than it needs for performing the necessary health checks, the concrete usage of the resource by the borrower is irrelevant for this purpose.

As a counter-example consider that instead of loaning the resource the Resource Pool Interface would completely encapsulate and hide the worker nodes, forcing the Execution component to go through it for every request that it wants to make—we will discuss this angle in detail later in this chapter as the Resource Pool Pattern. This would entail enabling the Resource Pool Interface to speak the whole language of a Worker Node in addition to its own. By loaning the resource the borrower may use it in any way necessary, but unbeknownst to the lender who is freed from the burden of having to understand this interaction. Consider the following possible conversation for a job execution:

- Execution sends job description to Worker Node
- Worker Node acknowledges receipt and starts sending regular execution metrics
- Execution might ask for intermediate results on behalf of end user (e.g. for a live logfile viewer)
- Worker Node replies with intermediate results when asked
- Worker Node signals job completion when done

- Execution acknowledges receipt and relieves Worker Node from this job

The individual messages that are exchanged are but small building blocks from which this interchange is composed and the purpose of the Resource Loan Pattern is to allow the lender to be largely unaware of this protocol that is only shared by the borrower and the resource as depicted in figure 13.1.

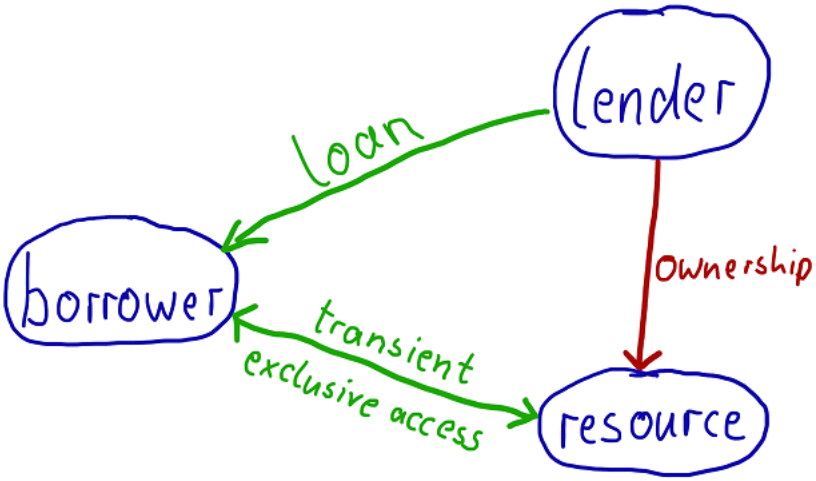


**Figure 13.1 The relationship between lender, borrower and resource in the loan pattern: the goal is to facilitate efficient exchange between borrower and resource while placing the burden of ownership with the lender.**

### *13.2.4 Applicability*

This pattern is applicable wherever a resource needs to be used by a component whose genuine responsibility does not a priori include the monitoring and lifecycle management of that resource. If the aspects of provisioning, monitoring and disposal can be factored out into their own component then the resource user is effectively freed from the details of these concerns, it is not forced to take on this incidental responsibility.

When deciding this question it is important to require the resulting resource manager component to be non-trivial, factoring out the management of a trivial resource only leads to additional runtime and design overhead, every component that is split out is to be considered as having a basic cost that needs to be offset by the benefits of the achieved decoupling and isolation.

### 13.2.5 Implementation Considerations

In the example that we considered the Execution component was in full control of the worker nodes, the question of how long to use them or when to shut them down was decided at its sole discretion. This would need to change if we assume that the underlying computing resource is indeed scarce and may need to be vacated in response to external events (for example when the per-minute runtime cost rises above a certain threshold). The Execution component would in this case only formulate the desired number of worker nodes and the decision of how many are provisioned is taken by the Resource Pool Interface. We might also envision that worker nodes may be reallocated to different Execution components for separate compute clusters.

This scenario requires that the resource lender retains the power to forcefully take back the resource when needed. If it hands out a direct reference to the loaned resource to the borrower then this is hardly possible: the borrower could just hold on to that reference and keep using it after the loan has been withdrawn. The solution is to hand out a proxy instead of the resource itself. This is easily possible in a setting where service references are location transparent since the borrower cannot care or know about the precise routing of requests to the resource. The proxy itself must be able to forward requests and responses between the borrower and the resource and it must also obey a deactivation command from the lender after which it rejects all requests from the borrower. In this fashion the lender can cut the resource loose from the borrower as required and decommission or reallocate the resource without interference from unruly borrowers.

Another consideration is that a resource that has been loaned to another service instance should be returned when the borrower terminates, otherwise the lender may not ever notice that the resource is not being used anymore—it might stay around perfectly healthy and fully functional for a very long time. Failing to recognize this situation amounts to a resource leak.

## 13.3 The Complex Command Pattern

*«Send compound instructions to the resource to avoid excessive network usage.»*

We have encapsulated the resources our system uses within components that manage, represent, or directly implement their functionality. This allows us to confine responsibility, not only for code modularity reasons (chapter 6) but also for vertical and horizontal scalability (chapters 4 & 5) and principled failure handling (chapter 7). The price for all these advantages is that we introduce a boundary between the resource and the rest of the system that can only be crossed by asynchronous messaging. The Resource Loan Pattern may help to move the resource as close as possible to its users but at least this one barrier will remain, leading to an increased latency and usually also decreased bandwidth for the communication between them. The core of the Complex Command Pattern lies in sending the behavior to the resource in order to save time and network

bandwidth in case of loquacious interchange between the two while the user of the resource is only interested in the comparatively small result.

This pattern has been used for this purpose since a long time, we approach it by way of a currently common example. Consider a large data set, so large that it cannot possibly fit into the working memory of a single machine. The data will be stored on a cluster of machines, each having only a small fraction of it—we call this Big Data. This data set is a resource that will be used by other parts of our system. Other components that interact with these data will send queries that have to be dispatched to the right cluster nodes according to where the data are located. If the data store only allowed the retrieval of individual data elements and left it to the clients to analyze them then any summarizing operation would involve the transfer of a huge amount of data; the resulting increase in network usage and the correspondingly high response latency are exacerbated by more complex analyses that require a frequent back and forth between the client and the data source. Therefore Big Data systems work by having the user send the computation job to the cluster instead of having it interrogate the cluster from the outside.

Another way to think of this pattern is by picturing the client and the resource (e.g. the large data set) as two nations that are about to negotiate a contract. To facilitate an efficient exchange an ambassador (the batch job) is sent from one nation to the other, negotiations may take many days but in the end the ambassador comes back home with the result.

### 13.3.1 The Problem Setting

We can generalize this problem as follows: a client wants to extract a result from a resource, a value that is relatively small compared to the amount of data that would need to be moved in the course of the loquacious exchange that is needed to obtain that value. The computation process is more intimately coupled with the data than with the client component that initiates it, the client is not genuinely interested in how the data are processed as long as it gets its result back. The resource, on the other hand, only holds the data and does not know the process by which the client’s requested result can be extracted. This means that the process description needs to be sent from the client to the resource, the client needs to have been provided with this description by the programmer who presumably knows both the need and the structure of the data.

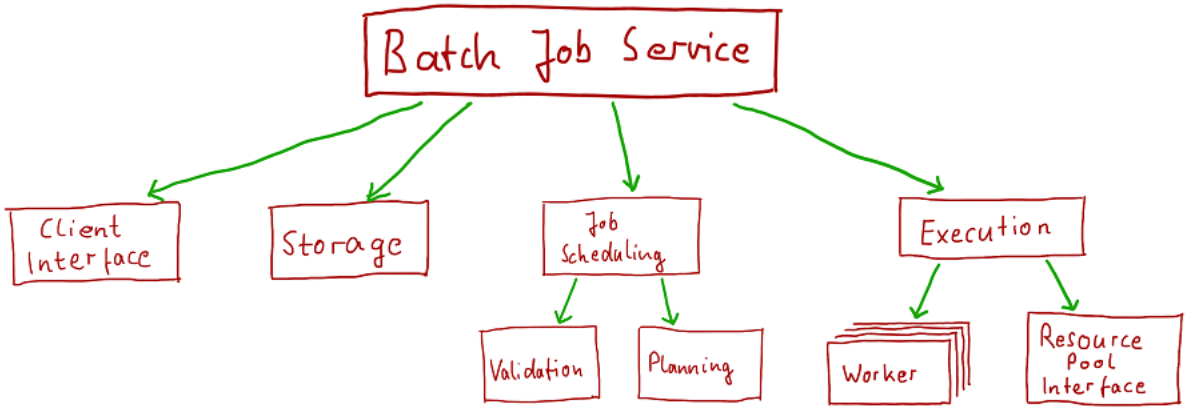


**Figure 13.2 The component hierarchy of the batch job service as derived in section 12.2.**

This is precisely what a Batch Job Service is all about, we just amend the mental picture slightly (the graphical overview does not change from section 12.2 but we brought forward figure 13.2 to refresh it in your mind). The worker nodes are no longer stateless services that can be provisioned and decommissioned dynamically but instead there is a fixed set of worker nodes in the Big Data cluster where each persistently holds the partition of data it has been entrusted with. The Execution component then will take care to send the jobs to the right worker nodes according to the data they need, which in turn will have an influence on how the scheduling decisions are made. These consequences—while interesting—are highly dependent on the particular example that we have chosen, more illuminating in terms of the generic pattern is the question of what exactly constitutes are Batch Job and how it is executed by a worker node. We will see in the following that there is more than one answer to this.

### *13.3.2 Applying the Pattern*

We start by considering what the essential pieces are that need to be conveyed. In order to route the job to the right nodes we need to know which data are going to be needed, the data set descriptor will thus need to be part of the batch job definition. The Scheduler will have to inspect this information and factor it into its decisions, which is to say that it needs to only read and understand this part. The Execution component on the other hand will have to split up the job into pieces according to the partitioning of the data itself, in addition to reading and understanding the data set descriptor it will need to be able to create copies of the overall batch job that act on subsets of the data, to be executed by the individual worker nodes.

Another approach would be to always send the full job description and have the worker node ignore all parts of the data set that it does not have, but this would be problematic in case the distribution of data changes or data partitions are replicated for fault tolerance: without making the data selection consistently at one place it would be hard or impossible to guarantee that in the end every requested data element has been processed exactly once. Sending the narrowed-down selection to the worker nodes gives

them an unambiguous instruction and allows them to signal that part of the requested data are no longer available at their location.

This leads us to the conclusion that a batch job must be a data structure that can be constructed not only by the client but also by the batch service's components. The first part of the data that it contains is a data set descriptor that several batch service components will need to understand and possibly split up.

The second piece of information that must be conveyed is the processing logic that acts upon the selected data. For this purpose the batch job must describe how this logic consumes the elements of the data set and how the resulting value is produced in the process. The last missing piece is then a recipe for combining the partial results from the individual worker nodes into the overall result that is shipped back to the external client.

### USING THE PLATFORM'S SERIALIZATION CAPABILITIES

Neglecting all incidental information like client authentication, authorization, quota and priority management etc. the essential structure of a batch job is captured in the following class definitions:

```
public interface ProcessingLogic {
  public PartialResult process(Stream<DataElement> input);
}

public interface MergeLogic {
  public Result merge(Collection<PartialResult> partialResults);
}

public class BatchJob {
  public final String dataSelector;
  public final ProcessingLogic processingLogic;
  public final MergeLogic mergeLogic;

  public BatchJob(String dataSelector,
                  ProcessingLogic processingLogic,
                  MergeLogic mergeLogic) {
    this.dataSelector = dataSelector;
    this.processingLogic = processingLogic;
    this.mergeLogic = mergeLogic;
  }

  public BatchJob withDataSelector(String selector) {
    return new BatchJob(selector, processingLogic, mergeLogic);
  }
}
```

The data selector is assumed to have a String-based syntax for the sake of simplicity—describing data sets is not the primary focus of this pattern—and a copy constructor is provided by way of the withDataSelector() method so that the Execution component can derive jobs that act on subsets of the data.

The more interesting piece that we will now examine in greater detail is the logic that is conveyed, represented here as two interfaces for which the client will need to provide implementations. The ProcessingLogic describes how to compute a partial result from a

data set that is represented as a stream[128] of elements: we are dealing with potentially Big Data that do not fit into memory all at once, hence passing the full Collection<DataElement> into the processing logic could easily lead to fatal working memory exhaustion (i.e. an OutOfMemoryError in Java). The MergeLogic then takes the partial results and combines them into the overall result that the client wants; here we expect the involved amount of data to be relatively small—even thousands of partial results should not take up a large amount of memory since we are working under the assumption that the result value is much smaller than the analyzed data set.

Footnote 128 see https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

In order to send a BatchJob message from the client to the Batch Job Service we will need to serialize this Java object. In the code above we could add “extends Serializable” in a few places and add some serialVersionUID values with the result that the Java Runtime would be able to turn a BatchJob object into a sequence of bytes that can be transferred. On the other end—within the Batch Job Service—we would need to reverse that process but here we hit a snag: the Java Runtime can only deserialize those classes whose definition it already knows about. The serialized representation contains only the class names of the objects that are referenced and the serialized form of the primitive values that they contain (like integers, characters, arrays), what is missing is the byte-code that describes the behavior of the objects.

In order to transfer that we will have to add the corresponding vocabulary to the protocol between the batch service and its clients, they will have to upload the JAR[129] files in conjunction with the job itself so that the necessary class definitions can be made known wherever a BatchJob message needs to be interpreted. This can be a tedious and brittle undertaking where any forgotten class or wrong library version leads to fatal JVM errors that make it impossible to run the job. It should also be noted that this approach ties both clients and service together in their choice of runtime environment: in the above example both parties need to use compatible versions of the Java runtime in order to successfully transfer and run the byte-code as well as the serialized objects, using the batch job service from a client written in JavaScript, Ruby, Haskell, etc. would not be possible.

Footnote 129 Java Archive, basically a ZIP-compressed file containing the machine-readable class definitions of a library, organized in class files.

### USING ANOTHER LANGUAGE AS BEHAVIOR TRANSFOR FORMAT

Another way to look at this is that the batch job service defines a language choice that is implicit to its client interface protocol. Batch jobs must be formulated such that the service can understand and execute them. We can turn this around to exercise greater freedom in this regard: if the service—still written in Java—were to accept the processing logic in another language, preferably one that is widely used and optimized for being shipped around and run in a variety of environments, then we could sidestep that rather tight code coupling that we faced when transferring Java classes directly. There are several options in this regard, examples being JavaScript due to its ubiquity and ease of interpretation or Python due to its popularity for data analytics purposes.

The Java 8 runtime includes a JavaScript engine which we can readily use to demonstrate this approach:

```
public class PartSuccess implements PartialResult {
  public final int value;
  public PartSuccess(int value) { this.value = value; }
}

public class PartFailure implements PartialResult {
  public final Throwable failure;
  public PartFailure(Throwable failure) { this.failure = failure;         }
}

public class BatchJobJS {
  public final String dataSelector;
  public final String processingLogic;
  public final String mergeLogic;

  public BatchJobJS(String dataSelector,
                    String processingLogic,
                    String mergeLogic) {
    this.dataSelector = dataSelector;
    this.processingLogic = processingLogic;
    this.mergeLogic = mergeLogic;
  }

  public BatchJobJS withDataSelector(String selector) {
    return new BatchJobJS(selector, processingLogic, mergeLogic);
  }
}

public class WorkerJS {

  public PartialResult runJob(BatchJobJS job) {
    ScriptEngine engine = new ScriptEngineManager()
                              .getEngineByName("nashorn");
    Invocable invocable = (Invocable) engine;
    try {
      engine.eval(job.processingLogic);
      final Stream<DataElement> input = provideData(job.dataSelector);
      PartialResult result =
       (PartialResult) invocable.invokeFunction("process", input);
      return result;
    } catch (Exception e) {
      return new PartFailure(e);
    }
  }

  private Stream<DataElement> provideData(String selector) {
```

```
    /* fetch data from persistent storage in streaming fashion */
  }
}
```

The processing logic is passed as a trivially serializable String that contains a JavaScript text which will define a process() when it is evaluated. This function is then invoked with the stream of data elements and expects a Result back. A very simple example of a processing logic script could look like this:

```
var PartSuccess = Java.type(
    'com.reactivedesignpatterns.chapter13.ComplexCommand.PartSuccess');

var process = function(input) {
  // 'input' is a Java 8 Stream
  var value = input.count();
  return new PartSuccess(value);
}
```

The code of these snippets is available in the source code archives on github if you want to play around with embedding JavaScript parts in your Java applications.

To recapitulate, we have explored two ways of transferring behavior—the processing logic—from client to batch service, one tied to the Java language and runtime and one in terms of a different language. The latter is used as a behavior exchange format that is possibly foreign to both parties but has the advantage of being easily transferable and interpretable. What we have not considered so far are the security implications of letting clients submit arbitrary processing instructions to our Big Data cluster—while they are meant to only analyze heaps of data they are in fact capable of calling any public method in the Java runtime environment, including file system access etc.

In order to secure our cluster we could implement filters that inspect the submitted code (hairy to get right in terms of not rejecting too many legitimate jobs), we could restrict the script interpreter (hairy in terms of not rejecting all malicious jobs), or we could use a behavior exchange format that can only express those operations that we want to expose to clients. Only the last option can deliver in terms of security and safety, but the price is rather high because most readily available languages are intended for general purpose use and therefore too powerful.

### USING A DOMAIN SPECIFIC LANGUAGE

*The following section describes techniques that are very powerful but require deeper knowledge and greater skill than is available to beginners. If you do not fully understand how the presented solutions could work you may still store away their features as an inspiration for what is possible.*

Pursuing the first two options is very specific to the example that we have chosen and leads us to acquire an intimate knowledge of JavaScript but it is the third option that is of more general value. What we need to do to go down this path is to devise a *domain*

*specific language*, in short a DSL. As Debasish Ghosh discusses in «DSLs in Action»[130] there are two basic forms of such languages:

Footnote 130 Manning Publications, 2011

- an *internal DSL* is embedded and expressed in the syntax of a host language
- an *external DSL* stands on its own

Designing an external DSL involves the creation of its grammar, the implementation of a corresponding parser in the language from which the DSL shall be used, and typically also some tooling to validate or otherwise automatically process documents in this language. The advantage of an external DSL is that it is not encumbered by the syntactic rules of a host language, it can be designed with complete freedom. We could imagine a Big Data language that describes the stepwise treatment of the input like so:

```
// give names to the data records' fields as needed
FOREACH Car (_, _, year, price)
SELECT year  1950 && year < 1960
MEDIAN OF price
REMEMBER AS p

// in the second iteration
FOREACH Car (make, model, _, price)
SELECT price > p
DISTINCT VALUES OF (make, model)
RETURN AS RESULT
```

Evaluating this script would iterate over the data set twice, first to find the median price for cars from the 1950's and then to collect all those pairs of make and model which cost more than that. The text shown above would be the serialized form of the program and by restricting the commands that are allowed we can exercise tight control over what client code can do. The worker node parses such scripts into a syntax tree and either interprets that one directly or compiles it into an executable program for the platform it is running on—in our Java example this would typically mean emitting byte-code that corresponds to the process described in the DSL. The latter is only needed when the DSL includes constructs like loops, conditional and recursion that lead to a high volume of interpreted expressions; in the example above the interpretation of the given statements would like consume a negligibly small amount of CPU cycles compared to the actual computation being carried out over a large data set[131].

Footnote 131 The only part that would need to be interpreted for each data element is the formula that is used for selecting cars by year, everything else lends itself well to being offered as a prepackaged compound operation.

If complete freedom of expression is not of primary importance an internal DSL might be a better fit—constructing and maintaining a parser and interpreter for a custom

language adds considerable development effort and organizational overhead, not to mention that designing a language that is reasonably self-consistent is an talent that not every engineer is gifted with.

As an existing example of an internal stream processing DSL consider the Akka Streams library: the user first creates a Graph—an immutable and reusable blueprint for the intended processing topology—and materializes that in a second step to be executed, typically by way of a group of Actors. We can separate these two steps so that the Graph is created at the client, serialized and submitted to the batch service, and finally deserialized and executed on the worker node. Defining a Graph corresponding to the external DSL example could look like this:

```
RunnableGraph<Future<Long>> p =
  Source.<DataElement>empty() // representing the real data source
    .filter(new InRange("year", 1950, 1960))
    .toMat(Sink.fold(0L, new Median<Long>("price")),
           Keep.<BoxedUnit, Long>right());

Source.<DataElement>empty()
  .map(new Inject<Long>(p, "p"))
  .filter(new Filter("price > p"))
  .to(Sink.fold(Collections.emptySet(),
                new DistinctValues<Pair<String, String>>("make", "model")));
```

Normally the map, filter and fold operations would accept any function literal (lambda expression) that we provide and syntactically that would still be valid here as well. Using arbitrary code would bring us back to the problem of having to transfer the user's byte-code to the batch service, though, which is why we provide a restricted vocabulary that is guaranteed to be known at the batch service's worker nodes. The operations that we offer can be compound ones like the median calculation above: we placed this element in a data sink that folds the incoming elements with the provided function, starting at the initial value that is given a 0L here. What happens behind the scenes is that the Graph layout is just recorded with the objects that we provide so that we can inspect the Graph in order to serialize it—when we encounter the Median object in the position of a folding function we know that we can transfer this behavior to the worker node, the only information that we serialize in addition to the operation name is the field name for which the median is to be calculated. You can see a sketch of the necessary class definitions in the source archives on github.

The same principle applies to the filtering steps, where we might have prepackaged operations like InRange that are configured with a field name, a minimal, and a maximal permissible value. We can also combine this approach with an external DSL as shown in the case of the generic Filter operation; implementing a parser and interpreter for simple mathematical expressions is not as complex as for a full-featured language and is general enough to be reusable across projects.

The approach shown above works best if the Akka Streams library is present on both ends which saves us the effort of creating the basic Graph DSL infrastructure and the stream processing engine, we just have to provide the specific operations that shall be supported. If more flexibility is needed then the serialization format that is chosen for these Graphs[132] can serve as an external DSL for client implementations that are not based on the JVM or want to use a different code representation for the processing logic.

Footnote 132 Akka Streams itself does not offer serialization of Graphs at the time of writing (version 1.0).

### 13.3.3 The Pattern Revisited

We started from the premise of sending the behavior to the resource in order to save time and network bandwidth in case of loquacious interchange between the two while the user of the resource is only interested in the comparatively small result. Exploring the possibilities we found several solutions to this problem:

- If user and resource are programmed for the same execution environment then we can write the behavior directly in the same language as the user code and send it off to the resource. Depending on the choice of execution environment this can incur considerable incidental complexity—in the case of Java classes we would for example need to identify all required byte-code, transfer it and load it at the receiving end—and it should also be noted that the choice will be hard to revert later because the implied behavior exchange format is coupled tightly to the runtime environment.
- To overcome the limitations of directly using the host language we can choose a different language as our behavior transfer format, picking one that is optimized for being transferred to remote systems and executed there. We looked at JavaScript as an ubiquitous example of this kind that is also supported directly by the Java runtime since version 8.
- If security is a concern then both previous solution suffer from being too expressive and giving the user too much power, the resource would execute foreign behavior that can do absolutely anything on its behalf. The best way to secure this process is to restrict what users can express by creating a domain specific language. This can be either an external one—with full freedom in its design but correspondingly high cost—or an internal one that reuses a host language or even another internal DSL as shown on the basis of the Akka Streams library.

In the example we considered there was a second piece of information that needs to be conveyed from the user to the resource, namely selecting the data set that the batch job shall process. This is not a general characteristic of this pattern, the behavior that is sent to the resource may well have all the power to select the target of its operations, such routing information will typically only be relevant within the implementation of the resource and in case of a DSL-based behavior description it will usually be possible to extract the needed selectors from the serialized behavior.

### 13.3.4 Applicability

The Complex Command Pattern provides decoupling of user and resource: the resource can be implemented to support only primitive operations while the user can still send complex command sequences in order to avoid sending large results and requests over the network in the course of a single process. The price for this is the definition and implementation of a behavior transfer language. This will have a high cost in terms of development effort independent of whether we use the host language and make it fit for network transfer, choose a different language, or create a domain specific language—particular care is needed to secure the solution against malicious users where required.

The applicability of this pattern is therefore limited by the balance between how valuable the achieved decoupling and network bandwidth reduction is in the context of the project requirements at hand. If the cost outweighs the benefits then you need to pick one:

- provide only primitive operations to make the resource implementation independent of its usage, at the cost of more network round-trips
- implement those compound operations that are needed by the clients within the protocol of the resource to obviate the need for a flexible behavior transport mechanism

The first option values code modularity over network usage, the second does the reverse.

## 13.4 The Resource Pool Pattern

*«Hide an elastic pool of resources behind their owner.»*

So far we have discussed the modeling and operation of a single resource as well as its interaction with its clients. The astute reader will have noticed that there is something missing from the full picture: the core principles of reactive system design demand replication. Recalling the discussion in chapter 2 we know that resilience cannot be achieved without distributing the solution across all failure axes—software, hardware, human—and we know that elasticity requires the ability to distribute the processing load across a number of resources that is dynamically adjusted to the incoming demand.

In the previous chapter we learnt about different ways to replicate a component, the effort put into this mechanism depends vastly on in how far the component’s state needs to be synchronized between replicas. This pattern focuses on the management and external presentation of the replicas. In keeping with the reasoning presented in chapter 4 & 5 it relies heavily on asynchronous message passing and location transparency in order to achieve scalability.

### 13.4.1 The Problem Setting

The example that readily offers itself is the batch job service that we have been building and enhancing throughout the previous sections. While the overall system implements a more complicated resource pool with sophisticated scheduling of complex commands (the batch jobs) we find a simple and pure example of a resource pool in the Execution component: after the Scheduler has decided the order of the next jobs to be run the Execution picks them up and assigns them to worker nodes as they become available—either by finishing their previous work or by being provisioned in response to rising demand.

Instead of investigating the details of how the relationship between Scheduler, Execution and worker nodes is represented in code we will focus on the messaging patterns that arise between these components, in particular around lifecycle events for worker nodes or the Execution component. This will illuminate their relationship in a fashion that is more easily applied to other use-cases.

### 13.4.2 Applying the Pattern

The part of the batch job service that we are looking at is the logic within the Execution component that distributes incoming batch jobs onto the available worker nodes; the jobs have previously been pulled from the schedule that is published by the Scheduler component. The basic process that we have assumed so far is shown in figure 13.3 using the conventions for diagramming reactive systems that are established in appendix A.

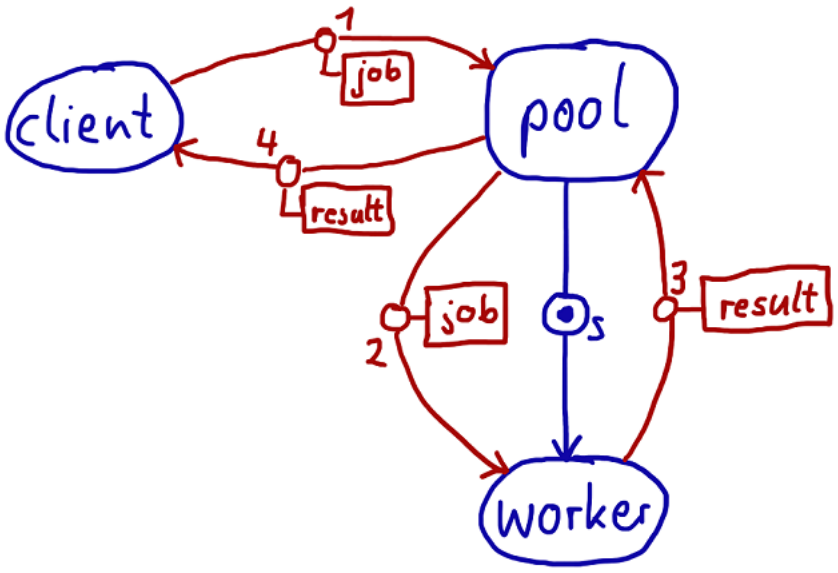


**Figure 13.3 The client represents the source of the batch jobs which is the part of the Execution component that has pulled the jobs from the published schedule. The pool is the sub-component that creates, owns and supervises the worker nodes. Neglecting that multiple workers may collaborate on one job the basic flow sends the job to the worker and conveys the result back to the client.**

Using this messaging topology the pool stays in control of all aspects of the workers' lifecycle: it sends the jobs, gets back the results, creates and terminates them—the pool always knows the current status of a worker node. Creating a new worker will typically happen in response to work being available, the process is depicted in figure 13.4:

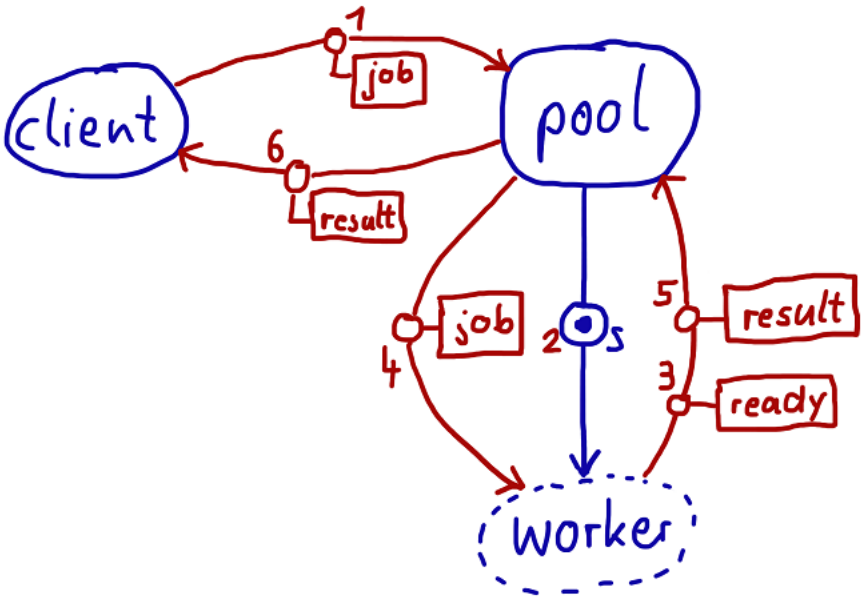


**Figure 13.4 Compared to the previous process we insert steps 2 and 3 in order to create the worker (using the infrastructure service as discussed in the Resource Encapsulation Pattern) and await its readiness.**

While this message flow represents the working principle it should not be taken too literally: the job that triggers the creation of the new worker node may well be handed to a different worker than the one being created, especially if it takes a long time to provision the worker node. The new worker then gets the next job that is to be dispatched after it has signaled readiness to the pool; in this sense readiness is the same as sending back a result for the implied job of getting started up.

During periods of reduced processing load the pool will notice that worker nodes are idle and since the pool knows how much work it distributes and which fraction of nodes are idle it is also in a good position to decide when to shut down a node. The corresponding message flow diagram is shown in figure 13.5.

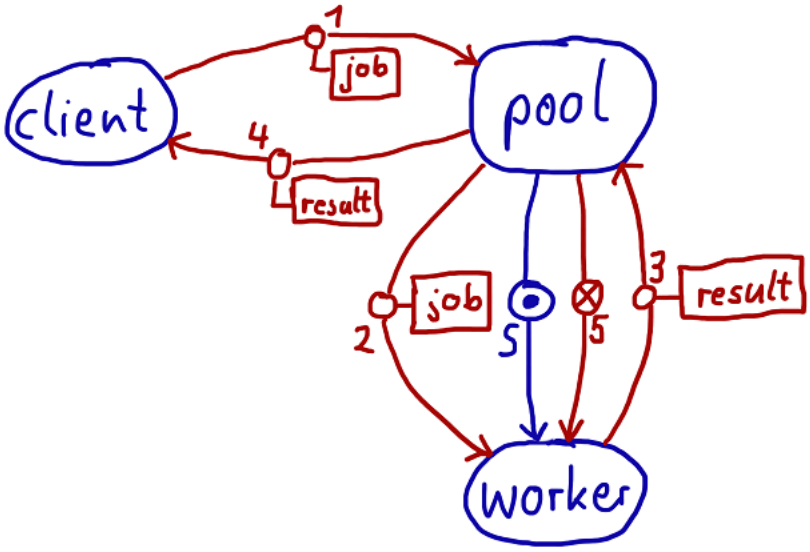


**Figure 13.5 After the worker has finished processing a job the pool sends the termination signal and refrains from sending further jobs to this worker. There can be a sizable delay between messages 4 and 5 in order to implement an idle timeout.**

The final aspect of the worker's lifecycle is how to handle failures. The pool should monitor all worker nodes by performing periodic health checks[133] and possibly also asking for progress updates. When a worker fails while processing a job the pool either sends this failure back to the Execution component to be recorded and relayed to the external client or it retries the job by giving it to another worker. The recovery process for the failed worker depends on the requirements of the use-case, the preferable approach is described in the Let-It-Crash pattern: just decommission the worker and all its resources and provision a fresh one. When a worker fails while being idle only the recovery process needs to be performed.

Footnote 133 This functionality might already be included in the services framework that is being used.

The message flows that we have seen so far all assume that the worker node receives a job within a single message and replies with a result that again fits within a single message. This covers a broad range of services but by far not all of them, notable exceptions being streaming services that provide a response that is a priori not bounded or where the purpose is the transmission at a given rate so that the external client can process the data as they arrive without having to buffer all of them. Another exception has been discussed in the Resource Loan Pattern in that the external client may reserve a worker and engage in an ongoing conversation with it to perform a complex task.

Accommodating these usage patterns requires a slight reinterpretation of the basic message flow as depicted in figure 13.6. Variants of this flow would send all messages between worker and external client (step 5) via the intermediary that represents the client for the pool, or signal completion not from the worker but instead have the external client convey this signal via the same route that is taken by the job description itself. The

former would allow tighter control over which protocols are permitted between external client and worker while the latter would give the pool more detailed information about when workers are finishing their jobs.

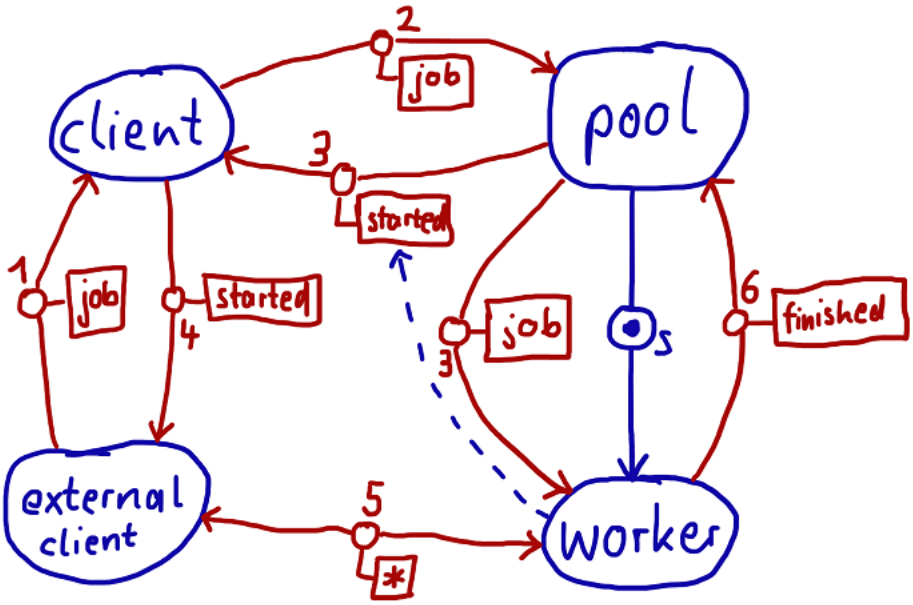


**Figure 13.6 In response to the job request message the pool will allocate a worker node and the inform both that node and the requester of the work to be done and the worker's identity, respectively. The external client can then engage directly with the worker until the job is finished, which the worker signals to the pool and thereby becomes eligible for further work.**

The important notion is that we retain the same processes for the creation and termination of worker nodes by keeping the same basic message flow structure between pool and worker: the pool initiates the work and the worker eventually replies with a completion notification.

### 13.4.3 The Pattern Revisited

To recapitulate, we have illuminated the relationship between a resource pool and the individual resources that it owns by sketching their primary message flows. A resource pool is free to allocate resources or dynamically adjust their number because it is in full control of the resources, concerning their lifecycle as well as their utilization. How the resources are used depends on the pattern that is applied by external clients:

- the basic model is that a request is sent to the pool and a resource is allocated for the duration of this request only
- in order to transfer a large volume of data or messages in the course of processing a single request the resource can be loaned to the external client for direct exclusive access; all aspects of the Resource Loan Pattern apply, including the possibility to employ proxies to enforce usage limitations
- if the purpose of a loquacious exchange is only the extraction of a relatively small result

value the Complex Command Pattern can be used to avoid the overhead of loaning the resource to the external client

Another consideration is that the basic message flow we discussed involves a full round-trip between resource and pool to signal the completion of a request and obtain the next one. This is the most precise and predictable model and it makes sense where the time to process a request is much larger than the network round-trip time. Under different circumstances it will be beneficial to use a buffering strategy where the pool keeps multiple requests in flight towards a single resource and the resource processes them one by one. The results that the resource sends back allow the pool to keep track of how many requests are currently outstanding and limit that number.

A drawback of the queueing solution is that sending a request to a resource that is not currently free means that processing may be deferred unpredictably, an SLA violation for one request then has a negative impact on some of the following requests as well. Queueing also means that in case of a failure multiple request may need to be dispatched for retry (if appropriate).

In the introduction to this pattern we cited elasticity as well as resilience as motivations for replicating a resource and so far we have only considered the former. The latter is more troublesome due to the ownership and supervision relationship between the pool and its resources: if the pool itself fails then the resources are orphaned. There are two ways to deal with this, either we replicate the pool including its resources—running complete setups in different data centers for example—or we replicate only the pool manager and transfer ownership of resources in order to realize a fail-over between them. Ownership transfer can only be initiated by the current owner if that is still functioning, otherwise each resource will have to monitor the pool it belongs to[134] and offer itself to a new owner upon receiving a failure notification for its parent[135]. For the replication of the pool manager itself any of the replication schemes discussed in the previous chapter can be used, picked according to the requirements of the use-case at hand.

Footnote 134 This should typically be implemented by the services framework in a generic fashion.

Footnote 135 Readers fluent in Akka will notice that reparenting is not supported for Actors, but we are talking about a pool of resource components in general and not Actors that encapsulate resources. The service abstraction that we are referring to here is a higher-level construct than the Actor Model.

### 13.4.4 Implementation Considerations

This pattern may be implemented by the services framework itself: deployment of a resource would include replication factors or performance metrics for dynamic scaling, and looking up that resource would result in a resource pool proxy that is interposed between client and resource implicitly. This works without central coordination of the pool proxies generated at the lookup site if the resource is either stateless (i.e. just offering computation functions) or using multiple-master replication. In these cases there could be one proxy per dependent service instance, removing any single point of failure. A potential difficulty with this is that it assumes that the resources themselves handle incoming requests from multiple sources. This can be challenging in the context of applying the Resource Loan pattern or the Complex Command pattern since requests may be delayed in a less controlled fashion than if there is a central authority that distributes them.

## 13.5 Patterns for Managed Blocking

*«Blocking a resource requires consideration and ownership.»*

When showing example code for how the Execution component of our exemplary batch job service would provision new worker nodes we have already encountered the situation that an API that we were using was designed such that it would block its calling thread. In the case of the Amazon Web Services API there would be ways around that, but this is not always the case. There are many libraries or frameworks that we may want to or have to use that do not offer the option of event-driven interaction, JDBC[136] is a well-known example that comes to mind. In order to use these APIs within a reactive system component special care needs to be taken in order to properly manage the resources that are implicitly seized, most notably the threads that are needed for their execution.

Footnote 136 Java Database Connectivity (http://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/), part of the Java platform, is a generic access layer for relational databases by which applications are decoupled from the specific database implementation that is used; it is a standard mechanism to provide data sources through dependency injection in application containers, usually driven by external deployment configuration files.

### 13.5.1 The Problem Setting

Consider a component that manages knowledge in a fashion that translates well into a relational database model—this might be a booking ledger, user and group management, the classical pet shop, etc. In our batch service example this might occur in the authentication and authorization service that the Client Interface component uses to decide whether a given request is legitimate or not. We could write this component from scratch, modeling every domain object as a persistent actor, or we could just reuse the enormous power that is conveniently available through off-the-shelf database management systems. In the absence of good reasons to the contrary it is preferable to reuse existing and working solutions, so we go ahead and use a JDBC driver within our implementation based on Java and Akka actors.

The problem that we are facing is that executing a database query may take an arbitrary amount of time: the database may be overloaded or failing, the query is not fully optimized due to a lack of indices or it might just be a very complex query over a large data set to begin with. If we execute a slow query within an Actor that runs on a shared thread pool then that thread will effectively be unavailable to the pool—and thereby to all other actors—until the result set has been communicated back. Other actors on the same thread pool may have scheduled timers that may be processed only with a large delay unless enough other threads are available; the bigger the thread pool the more such blocking actions it can tolerate at the same time, but threads are a finite resource on the Java virtual machine and that translates to a limited tolerance towards such blockers.

### 13.5.2 Applying the Pattern

The source of the problem is that a shared resource—the thread pool—is being used in a fashion that is violating the cooperative contract of the group of users. Actors are expected to process messages quickly and then give other actors a chance to run, this is the basis for their efficient thread sharing mechanism. Making a resource unavailable to others by seizing it means that the Actor that blocks the thread claims exclusive ownership, at least for the duration of the database query in our example. We have seen in the Resource Loan Pattern that exclusive access can be granted from the owner to another component but that must always happen explicitly, we must ask the owner for permission.

A thread pool usually does not have a mechanism for signaling that a given thread is being blocked exclusively for the currently running task[137]. If we cannot ask the owner of a shared thread for permission the logical conclusion is that we need to own a thread on which we can run the database query ourselves. This can be done by creating a private thread pool that is managed by the actor: now we can submit the blocking JDBC calls as tasks to this pool.

Footnote 137 An exception is the ForkJoinPool which can be informed using ForkJoinPool.managedBlock() (see https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.ManagedBlocker.html) but also in this case the management of the additionally created threads is limited.

```
public enum AccessRights {
  READ_JOB_STATUS, SUBMIT_JOB;
  public static final AccessRights[] EMPTY = new AccessRights[] {};
}

public class CheckAccess {
  public final String username;
  public final String credentials;
  public final AccessRights[] rights;
  public final ActorRef replyTo;

  public CheckAccess(String username, String credentials,
                     AccessRights[] rights, ActorRef replyTo) {
    this.username = username;
    this.credentials = credentials;
    this.rights = rights;
    this.replyTo = replyTo;
  }
}

public class CheckAccessResult {
  public final String username;
  public final String credentials;
  public final AccessRights[] rights;

  public CheckAccessResult(CheckAccess ca, AccessRights[] rights) {
    this.username = ca.username;
    this.credentials = ca.credentials;
    this.rights = rights;
  }
}

public class AccessService extends AbstractActor {
  private ExecutorService pool;

  public AccessService(DataSource db, int poolSize, int queueSize) {
    pool = new ThreadPoolExecutor(0, poolSize, 60, SECONDS,
                                  new LinkedBlockingDeque<>(queueSize));

    final ActorRef self = self();
    receive(ReceiveBuilder
      .match(CheckAccess.class, ca -> {
          try {
            pool.execute(() -> checkAccess(db, ca, self));
          } catch (RejectedExecutionException e) {
            ca.replyTo.tell(new CheckAccessResult(ca, AccessRights.EMPTY),
                            self);
        }})
      .build());
  }

  private static void checkAccess(DataSource db, CheckAccess ca,
                                  ActorRef self) {
    try (Connection conn = db.getConnection()) {
      final Statement stmt = conn.createStatement();
      final ResultSet result = stmt.executeQuery("<get access rights>");
      final List<AccessRights> rights = new LinkedList<>();
      while (result.next()) {
        rights.add(AccessRights.valueOf(result.getString(0)));
      }
      final AccessRights[] ar = rights.toArray(AccessRights.EMPTY);
      ca.replyTo.tell(new CheckAccessResult(ca, ar), self);
```

```
    } catch (Exception e) {
      ca.replyTo.tell(new CheckAccessResult(ca, AccessRights.EMPTY), self);
    }
    // the Connection is implicitly closed here as part of the try statement
  }
}
```

One peculiarity of JDBC connections is that in order to fully utilize the computation power of a database server we typically need to create multiple of them so that the server can work on several queries in parallel. For this reason we create a thread pool of the desired poolSize—one thread per database connection—and submit tasks to it as they come in without keeping track of the number of running queries. When the incoming load gets higher there will be a point where all threads in the pool are constantly active and since the number of threads equals the number of database connections all those will be active as well. At this point new requests will start queueing up, in this example we do not manage this queue explicitly within the actor but instead configure the thread pool with a queue of limited capacity. Tasks that cannot be executed right away because all threads are busy will be held in this queue until threads finish their current work and ask for more. In case a task is submitted while the queue is full its execution will be rejected and we use this mechanism to implement the bounded queueing behavior that is necessary to fulfill the Responsiveness of reactive components as discussed in section 2.2.5.

Since the required bounded queue is already implemented by the thread pool we are free in this example to send the response back to the client directly from the database query task, no interaction with the AccessService is needed on the way back. If the need arises to keep this actor in the loop then we will send the database result to the actor and have that sent the final response to the original client. Reasons for doing this might be that we need to explicitly manage the internal request queue—for prioritization, ability to cancel, or some such—or that the database result is only one of several inputs that are needed to compose the final response. Having the queries as well as responses go through the actor would in general allow it to comprehensively manage all aspects of this process: the only way to manage a unit effectively is to have exactly one person responsible and keep that person fully informed.

### *13.5.3 The Pattern Revisited*

What we have done in the above example is a process of multiple steps:

- First we noticed that our use-case required a resource that was not immediately obvious. Resources are often represented by objects in the programming language that are explicitly passed around or configured, but there are implicit resources that are always assumed and rarely considered explicitly—like threads or working memory—because they are usually shared such that the system works well in most scenarios, by convention. Whenever we run into one of the corner cases we will need to make these resources explicit in our system design.

- Having recognized the resources that we need to perform a certain function we modeled them explicitly: in the example these are the thread pool and the database connection pool. The thread pool is configured with a given number of threads and a maximal submission queue length in order to place an upper bound on the amount of auxiliary resources that are used by it. The same goes for the database connection pool whose configuration we left out of the example code snippet.
- In keeping with Resource Encapsulation we have placed the management of these resources within one component, represented by an actor. The lifecycle of the resources is contained within the lifecycle of the actor. The thread pool is created when starting and shut down when stopping. The database connections are managed individually on a per-request basis, taking care to obtain them such that they are released independently of how the processing ends. This is inspired by Let-It-Crash thinking in that shutting down the thread pool will release all associated resources, including the currently used database connections.
- The last step implements the Simple Component Pattern in that the responsibility of the component we are developing is to provide authorization information for users based on their credentials. To this end we need both resources and the logic that utilizes them, all bundled up in one component that does only its job, but does it in full.

While there are many APIs widely employed that make implicit use of hidden resources like the calling thread we will see this pattern being used to manage the blocking of threads, hence the name. The term “managed blocking” became first known to the authors by being used in the ForkJoinPool that was developed for Java 7 although it is mentioned in the .NET ecosystem several years earlier. The ForkJoinPool is not intended as a generic executor but rather as a targeted tool to help the parallelization of tasks that operate on large collections of data, splitting up work amongst the available CPUs and joining the partial results together, and the managed blocking support is needed to keep the CPUs busy while certain threads are stalled waiting for IO (from network or disk). Therefore entering a managed blocking section will roughly speaking just spawn a new worker thread that continues the number crunching while the current thread becomes temporarily inactive.

We generalize this pattern such that it is more widely applicable by considering any kind of resource that needs to be blocked as well as allowing for tailored management strategies of the blocked resource.

An example could be the explicit management of a segregated memory region that is split off from the main application heap in order to both place an upper bound on its size and to be able to explicitly control the access to this resource—using normal heap memory is unconstrained for all components in the same process and means that one misbehaving party can hamper the function of all of them. Another angle is that managed blocking is similar to the Resource Loan Pattern in the case of resources that are already encapsulated as reactive components.

### 13.5.4 Applicability

The first step of this pattern—the investigation of the use of hidden resources in ways that violate their sharing contract—should always be applied. Failure to recognize such a case will lead to scalability issues or application failures through resource exhaustion (e.g. OutOfMemoryError) as we have frequently encountered in consulting engagements.

On the other hand it is important to not get carried away and claim exclusive ownership of resources at a too fine-grained level or in cases that could live with having only shared access. This is particularly visible in the case of thread pools: the actor model is so successful in utilizing the machine's resources due to sharing the core pieces—CPUs and memory—in an event-driven activation scheme. Giving one thread to each actor would seem to give each of them exclusive access to a computing resource but not only does this incur orders of magnitude more overhead per actor, it also leads to significant contention on the underlying CPUs with all thread fighting for their chance to run.

Recognizing the usage of resources is as important as learning to qualify in which cases exclusion is needed and where sharing is appropriate—and under which rules. Unfortunately there is no simple rule to decide this.

## 13.6 Summary

This chapter was devoted entirely to the modeling and management of resources. Just as that term is very generic the patterns that are described also apply flexibly in a wide range of cases.

- First we discussed that each resource should be owned by one component that is fully responsible for its lifecycle. Ownership is exclusive—there can only be one owner—but resources may be used by a wide variety of clients within or without the reactive system that we are building, sometimes needing more direct access than is possible by always having the owner as an intermediary.
- The Resource Loan Pattern helps by bringing the resource as close to its client as possible, it allows the distance to be reduced to only one asynchronous message passing boundary for the purpose of transient exclusive access in order to perform a complex operation.
- The Complex Command Pattern turns this around by packaging up the client's part of a loquacious exchange such that it can be sent to the resource, like sending an ambassador to a foreign country.
- The Resource Pool Pattern tackles the concern of implementing elasticity and resilience by hiding the individually owned resources behind their manager; the resulting increase in distance between clients and resources can be mitigated by combining this pattern with the aforementioned two.
- Finally, we considered resource usage within the implementation of components, especially those that make use of libraries and frameworks that implicitly utilize resources that are otherwise shared in reactive systems, exemplified on the example of managing the blocking of threads in an otherwise event-driven system.

# A Diagramming Reactive Systems

The central aspect of designing reactive systems is to consider the message flows that occur within them. This appendix establishes a graphical notation that is used to depict message flows throughout the book.

## A.1 Defining the Stencils

In the table below a number (like 1 or 2) always represents an ordering constraint: if a component does something in response to an incoming event with number N then the number on the outgoing event(s) must be greater than N. This will usually not be single natural numbers but a variant of vector clocks, more on this below.


**Figure A.1 primordial component**

A component that was created before the depicted message flow starts.

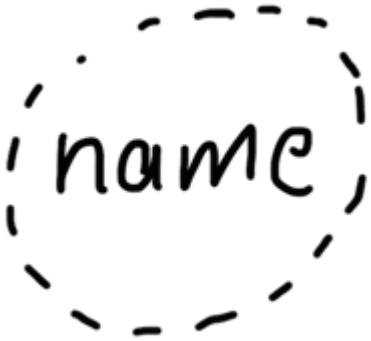


**Figure A.2 transient component**

A component that is created (and usually also destroyed) after the depicted message flow starts.

**Figure A.3 primordial creation**

A parent–child relationship: the component on the left initiates the creation of the component on the right.


**Figure A.4 creation with supervision**

A parent–child relationship with supervision: in addition to the above the child component’s failures are handled by the parent.

**Figure A.5 creation during the flow**

A parent–child relationship that begins with the creation of the child during the processing flow of the diagram (may be combined with supervision as well by adding the S).

**Figure A.6 termination command**

A termination command to a component, usually sent by its parent.

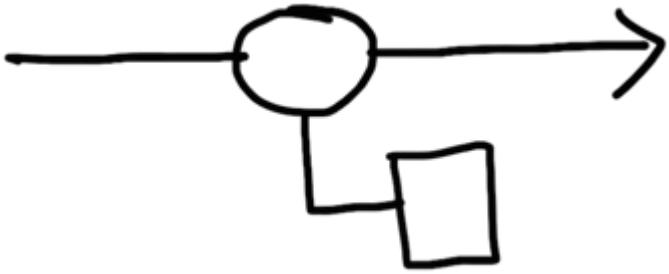

**Figure A.7 message**

A message sent from the component on the left to the component on the right. There are several variants of this where only the central piece is depicted below, but the arrows are implied for all of them.


**Figure A.8 message description**

A message send that is annotated with a description of the message contents. This variant is available for all kinds of message depictions shown below.

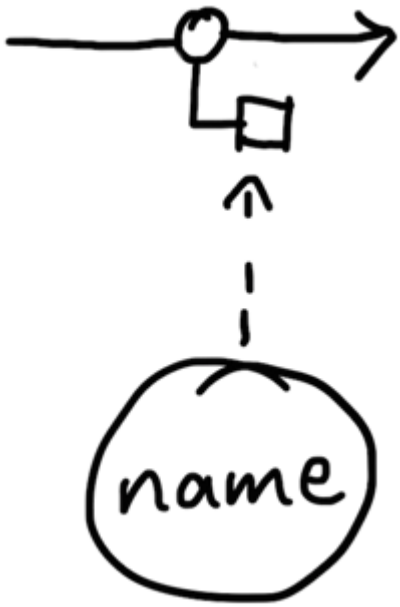


**Figure A.9 reference inclusion**

A message that includes the address of a component. The sender of this message must be in possession of this address when this message send occurs.


**Figure A.10 causality tracking number**

A message with its causality tracking number.

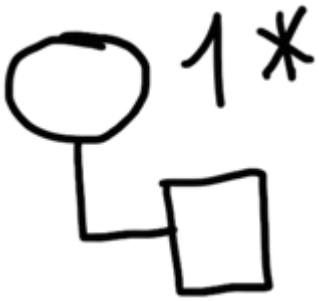


**Figure A.11 recurring message**

A recurring message send which begins at the indicated number.


**Figure A.12 scheduled message**

A message that is entered into the scheduler at number 1 and scheduled to be sent at number 2. The second number must always be greater than the first.

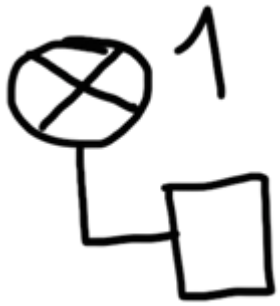

**Figure A.13 termination notification**

A termination notification message which is generated by the system to inform a component that another component has terminated (or is declared as such in case of a network partition).

**Figure A.14 sequence of messages**

A sequence of multiple messages sent one after the other, summarized with the same number.

**Figure A.15 failure notification**

A failure message sent from a component to its supervisor.