Module 1 Part 2: Reliable and Scalable Data-Intensive Applications

Introduction

Big Data involves storage and analysis of potentially massive data volumes and diverse data types. Modern organizations need people who can help implement the tools they need to deal with these huge data sets.

Learning Outcomes

- Identify the issues of scaling to Big Data
- Describe Reactive Design as a framework for high-quality data-intensive systems
- Discuss the impact of latency on user experience
- Contrast and recognize the importance of asynchrony and parallelism
- Explain strategies for highly reliable systems

Readings and Resources

We invite you to further supplement this notebook with the following recommended resource:

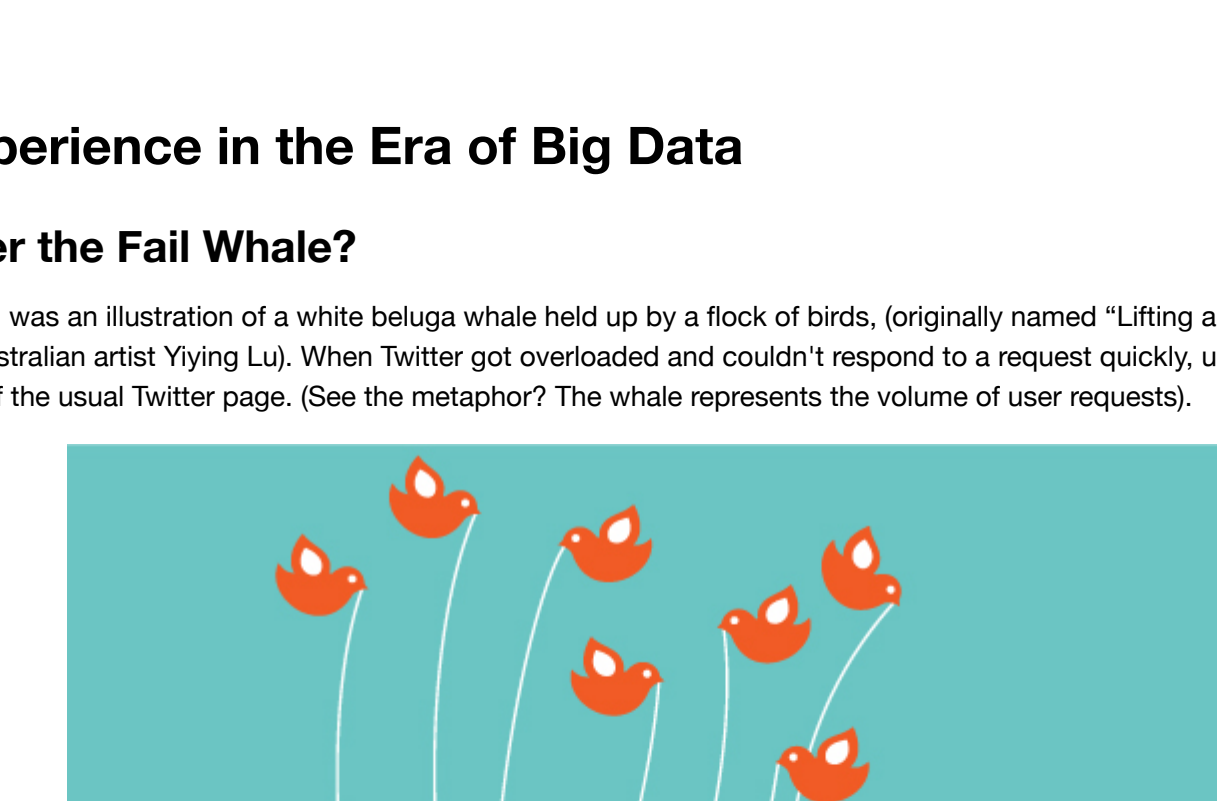
- Bonér, J., Farley, D., Kuhn, R. and Thompson M. (2014). Reactive Manifesto http://www.reactivemanifesto.org/

Table of Contents

User Experience in the Era of Big Data

Remember the Fail Whale?

The "Fail Whale" was an illustration of a white beluga whale held up by a flock of birds, (originally named "Lifting a Dreamer", and was illustrated by Australian artist Yiying Lu). When Twitter got overloaded and couldn't respond to a request quickly, users would see the Fail Whale instead of the usual Twitter page. (See the metaphor? The whale represents the volume of user requests).



Source: http://www.yiyinglu.com/?portfolio=lifting-a-dreamer-aka-twitter-fail-whale

Today, the Fail Whale no longer ever appears. This is because in the mid-2000's Twitter completely redesigned their systems using an approach called Reactive Design, which increases and decreases Twitter's cloud computing power in responwe to the amount of demand. We will look at Reactive Design in this module. It's a great way to think about large-scale data-intensive systems.

User experience expectations

We now live in a 7/24/365 world where Internet and system users have become accustomed to and expect fast, always-on electronic services. Google, Facebook and Amazon never have outages while they do backups or version updates. They very rarely have outages of any kind despite rolling out hundreds of changes to their production systems every day.

We also expect online services to be richly data-intensive. As an example, consider **Google Maps**:

- **Rich data sets**: Access to a vast quantity of highly-detailed data in a way that allows users to zoom in or out at will
- **Responsive**: Google Maps almost never locks up and shows a busy icon, nor are there periods when it is noticeably slower than other times
- **Highly Available**: It's always-on
- **Continuously Improving**: New features appear with little or no user action required to add them
- **Multi-platform**: Google Maps works on a wide variety of devices

Why is this difficult?

Making systems always-responsive like this is technically extremely challenging. Until recently, developers wrote systems with a simplifying assumption: they wrote the code as if the computer it was running on and the networks connecting to other resources were completely reliable. This is fine for a single-user or departmental application. Not so for Internet-scale.

In reality, many things can go wrong:

- Network links can get congested
- Network links can go down entirely
- Servers can go down or become unresponsive
- Programming errors can cause a sudden loss or flood of data
- Processes on a server can be temporarily blocked. Blocking means to suspend progress while waiting for something (e.g., applications written in languages that have built-in memory management such as Java can pause for a second or two with no notice to reclaim resources (so-called "garbage collection")
- Other services that a system calls upon to delegate a task could go down or take a long time to respond (e.g. a call out to a credit card processing service)
- The load may increase suddenly and dramatically as a result of many users simultaneously invoking the service
- Demand may ebb just as quickly, in which case costly computing resources should be freed for other tasks

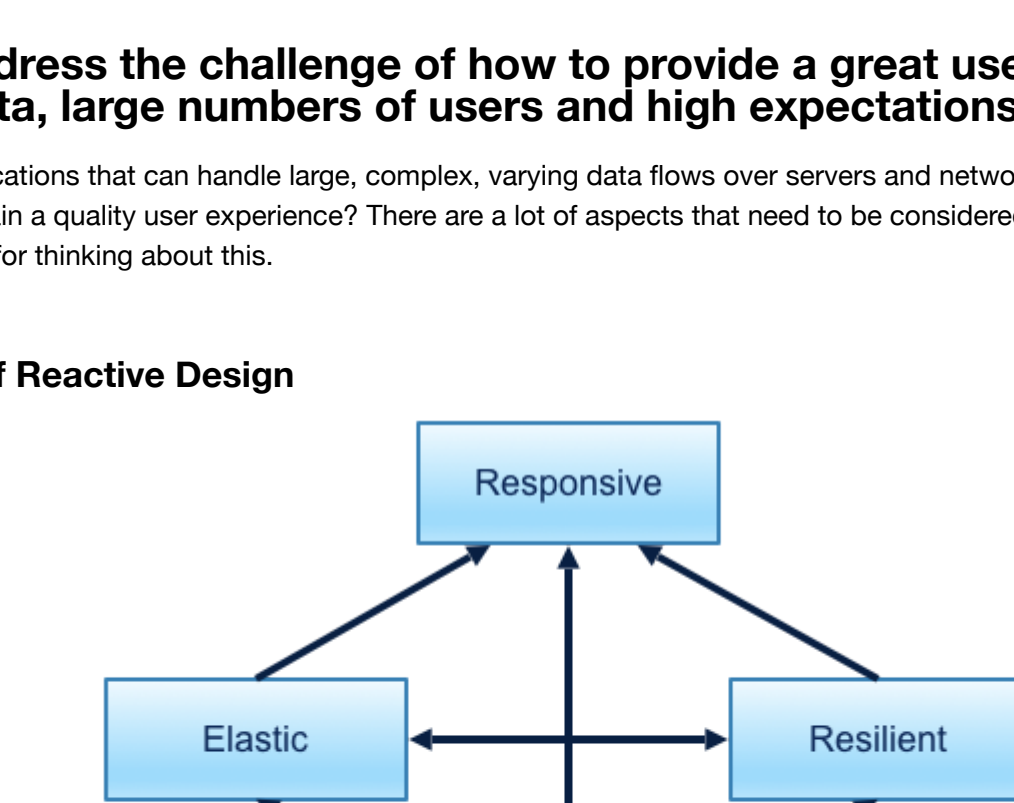
The new expectation is that all of these should be handled quickly and gracefully. A system should never lock up or show an hourglass for more than an instant. Accomplishing this requires a fundamentally different way of building systems. This isn't something that can be automatically handled for the application developer by the operating system and/or network software. Moving to "the cloud" doesn't fix it (but can be part of a solution). The right action to take when a service becomes slow or unavailable is often something that needs to be decided by the application, close to the end user, where the end user impact is best understood, not outside it in its environment somewhere.

Introduction to Reactive Design

How do we address the challenge of how to provide a great user experience in the era of huge data, large numbers of users and high expectations?

How do we create applications that can handle large, complex, varying data flows over servers and networks of limited and unreliable capacity, and still maintain a quality user experience? There are a lot of aspects that need to be considered and a variety of strategies. Let's begin with a framework for thinking about this.

The Four Pillars of Reactive Design



Reactive Design is a design philosophy which begins with the notion that systems should always be **responsive**, and to do that requires design for **elasticity** and **resilence**, which are best implemented by a **message-driven** architecture. Let's unpack what that means.

Responsiveness

- Systems should never "lock up" and should always provide continual feedback about progress on tasks that will take some time

Elasticity

- The system should stay responsive under varying workload
- Additional resources (e.g. servers, memory, bandwidth) should be made available when demand increases and released when not

Resilience

- The system should stay responsive even when failures occur
- This means recognizing that failure of hardware or software components is inevitable and that we must build the system from the ground up expecting occasional failures to occur

Message Driven

- Achieving responsiveness, resilience and elasticity requires a loosely-coupled design where individual components can fail but other units can recognize when a failure has occurred and work around or repair it
- Loose coupling of servers and services is best achieved by having them message each other rather than tying them together with tight dependencies that cause the entire system to fail if some part does

Responsiveness

A *responsive* system is one which never locks up, or **blocks**, (i.e., it is always live and interactive, even when doing a processing step that will take some time). This is especially critical for the user interface where there is a human directly involved. There are a lot of reasons that a system might become temporarily or permanently non-responsive: it's waiting for service invoked on another computer to complete, the server the system is running on hangs, etc. There are also reasons it might be convenient for a system designer to allow blocking, such as preventing a user from clicking a button more than once and ordering several pairs of sneakers by mistake. In terms of user experience, however, this leaves the user feeling like they've lost control. Has their browser broken? Will the task ever complete? Was my credit card charged but the product not ordered?

Blocking also has a negative impact on parallel processing. If a system process can't proceed because it's waiting for something, it should free up its CPU to do other useful work in the meantime. Otherwise the overall system throughput will suffer.

Strategies for responsiveness

- Systems should have a pre-established response time target that provides a consistent user experience
- Consistency is important: if an operation always takes about five seconds, then users will become comfortable with this; if it takes anywhere from 1 to 30 seconds, anxiety will set in after a few seconds
- Ideally the end user should never feel that the system they're using is "locked up" (or even sluggish). This means that control should be immediately returned to the user interface while other tasks are running.
- If operations will take some time the user should be informed in advance and updated continuously on progress
- If an operation appears to be unavailable the user should be informed immediately
- If possible the user should be offered an alternative if a service is not available, such as saving their work so far or continuing with a reduced service level of some kind

Designing a service for responsiveness

Let's consider the situation where we're not waiting for another service but instead other services are waiting for us. Perhaps our service is an online credit card fraud detector that runs a sophisticated inference using a previously-trained machine learning model.

- How should we handle a situation where requests arrive faster than we can process them? If we can find ourselves in a situation where its truly a flood of requests beyond what the provisioned servers can handle, we need to be able to scale up quickly. This is where modern elastic **cloud services** really shine.
- Even if the volume of requests is not so high that we can't handle it on average, there may still be spikes in demand just by normal variation. In this case, we need to use **queues** (in-memory buffers) to catch and save requests until they can be processed.
- Using queues will add some latency (delay) which needs to be taken into account, but it's usually better than to ignore requests we can't process immediately.
- Queues usually have a fixed size because they usually take up main memory and at a high enough level of load the server won't be able to keep up with demand anyways and would have to drop requests. If the buffer is getting full, it would be best to immediately alert the caller that the response will take longer than usual, and let the caller decide whether they want to wait or cancel the request. The application doing the call is closer to the end user and so is in a better position to judge whether waiting is appropriate or not, perhaps by letting the user know and asking whether they want to wait or abort. It would be even better if a waiting time estimate can be provided to allow the user to make an informed decision.

Elasticity

Elasticity and scalability

An elastic system is one which is rapidly scalable, both up and down. Let's begin by defining **scalability**.

"Scalability is the ability of a system, network, or process to handle a growing amount of work, or its potential to be enlarged to accommodate that growth. For example, a system is considered scalable if it is capable of increasing its total output under an increased load when resources (typically hardware) are added" (Scalability, nd).

NOTE: The term is also used to refer to the ability of a company to grow rapidly, but this is a different definition than we are concerned with here.

Throughput and Latency

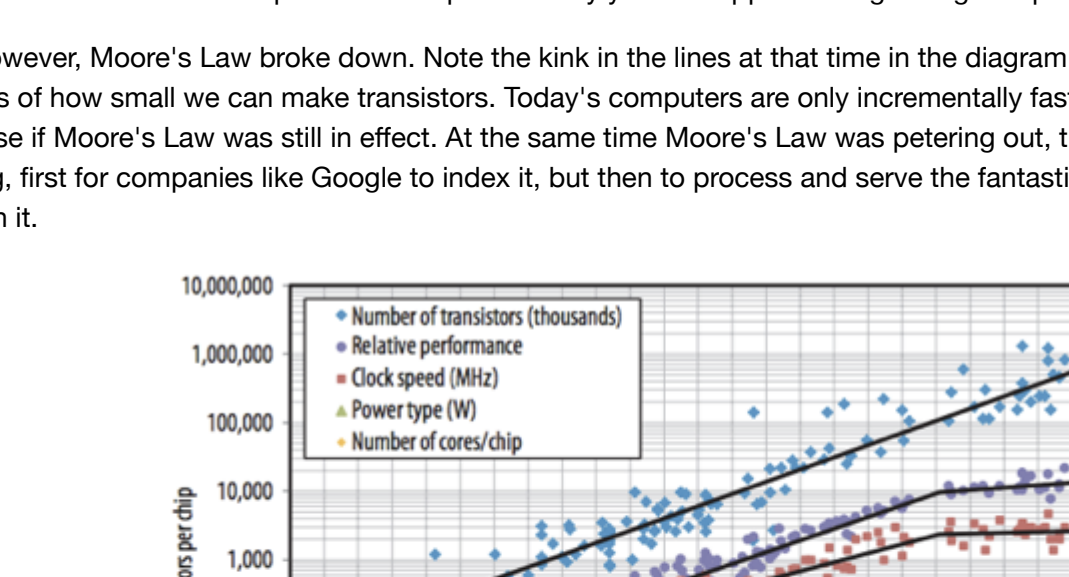
There are two important measures of scalability: **throughput** and **latency**. They are both important, and the strategies for improving them are different.

- **Throughput**: Throughput is the number of units of work the system can complete per unit time. For CPU's this is the number of units of computation that can be completed (for example, the number of similar transactions that can be processed). For networks, throughput is a measure of how much data can be transported each second. Consider the throughput required to support Twitter which handles tens of thousands of requests per second during peak times.
- **Latency**: The terms latency and response time are often used synonymously. Although there are some technical differences, it is what the user sees that matters: the time between when they hit a key or click a button and a response arrives. Delays can occur anywhere in a long chain of interactions across a network: at routers, over slow connections between routers, or during the actual processing of the request. Complex processing may require the participation of several servers in different locations, any of which could introduce delays. At Internet scale even the speed of light becomes a limiting factor: it takes an absolute minimum of 133 milliseconds (over a tenth of a second) for a message to travel through directly-connected fibre to a server on the other side of the world and back.

We're reaching the computer speed limit

The computer revolution has ridden the wave of Moore's Law since the 1970's. Gordon Moore, co-founder of Fairchild Semiconductor and long-time CEO of Intel predicted then that the number of transistors that could fit on a chip would double every 1-2 years which would allow the speed of computers to grow at about the same rate. Systems prior to the Internet typically resided on a single computer and to some extent relied on Moore's Law to provide more power every year to support their growing complexity and feature set.

Around 2004, however, Moore's Law broke down. Note the kink in the lines at that time in the diagram below. We're reaching the quantum mechanical limits of how small we can make transistors. Today's computer's are only incrementally faster than 2004's, not multiples as would be the case if Moore's Law was still in effect. At the same time Moore's Law was petering out, the Internet began demanded global-scale processing, first for companies like Google to index it, but then to process and serve the fantastic quantities of data that are now available through it.



Source: https://intelligence.org/2014/05/12/exponential-and-non-exponential/

Scaling out

CPU manufacturers have responded by designing chips with several CPU's on the same chip so the transistor count and total processing capacity has continued to increase, but those additional CPU's are only incrementally faster than 2004's, not multiples. This means that if we need more performance than a single CPU can provide we need to *scale out* across CPU's on a single machine. We can only do it though if the workload is amenable to being split into processes that can run in parallel. This introduces a *lot* of additional complexity if the parallel processes need to synchronize or share data, which they typically do. So although scaling "up" by using faster computers was once the dominant strategy, we no longer have a choice in many cases but to scale "out". If one machine's processing power still isn't enough, we then have to go even further and use *distributed processing* which as we will see complicates things even further.

Distributed computing

Today's cloud computing services provide the potential for rapid scaling, both up and down, in response to varying workloads. This is what we call *elasticity*: the ability to not only recruit additional processing nodes (servers) in a matter of minutes or even seconds when load spikes, but also to release computing resources back to a shared pool when no longer needed. It isn't as easy as just putting the processing in the cloud however. Cloud vendors provide solid tools for helping to build distributed systems, but there are a large number of concerns that need to be addressed by the system architect to ensure it operates correctly and meets the new user experience expectations.

At Internet scale, where we often have several online services running of different machines, possibly in different geographies, to satisfy a single request, it is impossible for the data to arrive at exactly the same time everywhere. Synchronizing data across these servers is a complex subject that takes careful consideration between unavoidable tradeoffs between correctness and timeliness. As we will discuss throughout this course, these issues forced a complete re-think of how engineers architect systems.

Predictive scaling

It takes time to recruit additional resources in the cloud. Adding a server may take a few minutes. That's a long time if demand is spiking. Staying responsive may require predictive algorithms that pre-scale in anticipation of rising demand before it actually occurs.

GPU's and tensor processors

For Machine Learning workloads, especially those involving Deep Learning, it may be necessary to use specialized processors. GPU's originally designed for gaming can be used to accelerate vector operations and most cloud vendors allow users to request servers with high-end GPU's for that purpose. Leading cloud vendors now also offer servers with specialized tensor processors designed specifically for Machine Learning work.

Strategies for reducing latency

The best way to reduce latency is to avoid blocking when waiting for something whenever possible.

A second strategy is to employ **parallelism**. For example, if the application has to call on and wait for responses from three other services over the network, do them in parallel if possible. The latency will then be the longest of the three response times rather than the sum.

If the program can't do anything useful until all of its requests return it could simply block. A better solution is to send the requests, then relinquish control of the CPU to other processes. Ideally we'd like to also unblock the user interface, at least offering a cancel and "estimated time to complete". This becomes increasingly easy for a programmer to do by using a feature appearing in a variety of programming frameworks called a **"Future"** or **"Promise"**. All such calls should have **timeouts** so if the request takes too long it is cancelled and control is returned to the user rather than locking up the browser.

It is also a good practice for the application to cancel the first request and retry it if a response doesn't arrive in a reasonable amount of time. (Note there may need to be a numbering scheme for the requests so when a reply does come back the application can tell whether it is a response to the first request or a retry).

Resilience

A system can't be scalable unless it's reliable. Everybody has an intuitive idea of what it means for something to be reliable or unreliable. For software, typical expectations include:

- The application does what was expected
- It catches user input errors and allows correction of mistakes
- Its performance is acceptable
- The system is secure
- If the system goes into some kind of error state it can reset itself and recover

For a system to be able to do all these things and also allow modifications to itself without downtime is a modern miracle. We call this kind of fault tolerance *resilience*.

Strategies for correctness

Let's focus first on strategies for increasing our confidence that when the system is operating as normal i.e. it's doing what we expect. There is a revolution in systems development currently going on around the following:

- Automated testing and Test-Driven Design
  - **Test-Driven Design (TDD)** is a discipline where tests are written at the same time as the code they test, and are used to verify correctness after each and every change to the system
- Static typing
  - **Dynamically typed** languages such as Python and JavaScript are relatively easy for beginners to work with but provide no feedback to the programmer about whether what has been written will run or abort when executed
  - **Statically typed** languages on the other hand will not compile and run at all unless the types used all match up
  - This can be frustrating for beginners as they can't experiment easily, but once the code compiles it is much more likely to be correct and reliable
- Static analysis
  - Tools such as linters can help to find common coding mistakes and enforce style standards which promotes code readability
- **Functional programming**, though older than the more common approach of imperative programming, is becoming increasingly popular because of the need to write programs to run on multiple CPU's
  - The attraction is that it is works most naturally with immutable values rather than variables
  - If values are immutable there can be no shared state between threads (code running in parallel) to be concerned about
  - Although the concepts in functional programming are new to most developers, and so the code can be foreign-looking at first, it is much easier to understand and reason about once mastered

Strategies for resilience

Assuming the code is correct, resilience in the face of the still-inevitable failures is achieved through:

- **Replication**: Replication of data, nodes and network connections so if a device or connection fails, others can be called on to fulfill the service request
- **Containment**: Failures are limited in scope and can't cascade to other devices
- **Isolation**: State and behavior are not shared between threads of parallel execution
- **Delegation**: Subtasks are delegated to other components that may run on a different thread/node

Resilience requires replication but this brings its own problems. If the copies are on different nodes, the delay can be significant. Consider a situation where two changes are made to a customer's address but they arrive in opposite orders at two different computers because of network delays. The two systems would record the latest change they received as the correct one and would no longer agree on the customer's address. Distributed systems must be designed to allow for, recognize and resolve these kinds of inconsistencies when they occur.

There will be much more on this topic in this course.

Resilience in biological systems

Think for a moment about how nature has "architected" life over 4.5 billion years of refinement to be robust despite an always-changing environment. Higher life forms are built of a multitude of components, at least some of which are redundant (e.g. kidneys, lungs), providing a measure of protection against failure or loss. Computer systems need to and are becoming more like biological systems in this sense.

Testing for Resilience

A new form of testing called *Chaos Testing* has become popular with Internet-scale service providers. Chaos testing means to intentionally cause failures such as shutting down servers at random, at first in a test environment, and then live, in production, to continually exercise and verify that the measures for resilience are working as they should.

Message Driven

Messages

So, our big data systems will need to be structured as colonies of cooperating, dispensible units. We need them to work together, but not be tied together so that if one submerges it drags others down with it. Let's look again at how nature has solved this problem: by having components communicate with each other: people by speech, animals by calls, cells by chemical messengers. The loss of any specific unit doesn't cause a cascade of other failures; the other units recognize that they're no longer receiving messages from the lost unit and adapt their behaviours accordingly.

Asynchronous, concurrent and parallel processing

The terms *asynchronous*, *concurrent* and *parallel* are related and sometimes used interchangeably, but there are differences between them when used properly:

- **Asynchronous**: An asynchronous system is one in which there is no central, coordinating clock. Execution of blocks of code occur in response to events (such as the arrival of a message from another unit) rather than in order according to the program's logic. Web servers must be at least somewhat asynchronous by their nature: they need to respond when a request arrives over the Internet, but may be idle otherwise. If not busy responding to requests at the moment, they should yield the CPU to other processes running on the server so they can proceed and do useful work.
- **Concurrent**: Concurrent systems can run several processes (programs/tasks) with overlapping lifetimes, either using true parallelism (we will define next) or by interleaving the processes so they share a single CPU. For example, Windows could run several programs at the same time before CPU's became multicore. It did this by letting each program run for a few milliseconds in a round robin arrangement, swapping between them quickly enough that it appeared as if they were all running simultaneously. Now that computers are essentially all multicore, some of the processes do actually run in parallel.
- **Parallel**: Parallel computing means that two or more processes are actually executing simultaneously on different CPU's, in the same computer or on different ones (in which case, as we mentioned, we usually call it distributed computing). In addition to processes/programs running in parallel, we can have parts of a single program run in parallel, in which case we call each of these parallel executions a **thread**. Modern systems run many more threads than they have CPU's.

Asynchronous Non-blocking Processing

Programming paradigms are in the midst of the most fundamental change since the introduction of object-oriented programming. Many popular programming languages are "blocking" i.e., the processor or virtual machine thread they are running on stops while waiting for something to happen e.g. I/O.

Switching to another thread is a computationally expensive process resulting in degradation of service, and if severe enough, results in refusal of new requests for service. New programming styles that yield control of a thread to other work while waiting are rapidly gaining favour. This allows threads to be reused rather than creating and destroying them.

Here are some of the most popular techniques:

- **Green threads**: This is an approach that is becoming popular in several languages where parallel processing is managed within the application itself rather than relying solely on operating system threads. This allows very efficient non-blocking use of threads but only works on a single computer. It cannot be used for distributed computing.
- **Communicating Sequential Processes (CSP)**: Supports message-passing between processes and is the basis of Go language's asynchronous non-blocking "channels" or "goroutines". Unfortunately this approach also does not scale to multiple machines.
- **Futures/Promises**: In this technique, a call for a resource returns immediately with a variable that will be "filled in" with a pointer to the resource as soon as it becomes available. Futures work with multiple CPU's but not multiple nodes. (A promise is a future where once the data arrives it becomes immutable).
- **Reactive Extensions**: Particularly useful for streaming data, a process registers a function with an event handler to be executed when a chunk of data arrives. This method can scale to multiple CPU's and nodes.
- **Actors**: This is the most sophisticated method and supports message passing between threads in addition to being non-blocking and asynchronous. Actors are blocks of code that only allow a single thread of execution and only communicate by passing queued, immutable messages. Actors were originally developed in 1973 by Carl Hewitt and were used to develop high-reliability phone switching systems but now are widely used in distributed systems.

Which to use?

The techniques above all have their place. If we are working on a single machine, Green Threads are likely a good choice and there are several libraries available that support them. Same with CSP for Go--it's built into the language. Python 3, JavaScript and Scala natively support Futures/Promises.

Reactive Design favours Actors because it is a single approach that can be used for parallelism on a single machine up to a highly distributed system. Robust actor implementations are only available for a few languages however, primarily Erlang, Java, Scala and .NET.

Microservices

Related to all of the above is the notion of microservices. A **microservice** is a like a tiny, single-purpose web application. It receives requests for service as http messages and sends its responses in the same way. If we design an application as a collection of microservices we will be effectively mirroring nature: they will be independent units that communicate with each other only through messages. Each individual microservice can even be programmed in a different programming language as the only way they interact with it is through http messages and so the services don't need to be linked together by a compiler. If a microservices architecture is done well, the services can be independently versioned, deployed and upgraded.

References

Abbott, M. & Fisher, M. (2015). *The Art of Scalability*, Addison-Wesley.

Bonér, J., Farley, D., Kuhn, R. & Thompson M. (2014). Reactive Manifesto http://www.reactivemanifesto.org/

Kleppmann, M. (2017). Chapter 1: Reliable, Scalable and Maintainable Applications in *Designing Data Intensive Applications*. O'Reilly: Boston. http://shop.oreilly.com/product/0636920032175.do

Kuhn, R., Hanafee, B. & Allen, J. (2017). *Reactive Design Patterns* Manning

Reactive Streams (n.d.). http://www.reactive-streams.org/

Scalability, (n.d.). In Wikipedia. Retrieved March 27, 2019, from https://en.wikipedia.org/wiki/Scalability