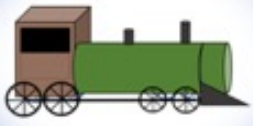
A History of Application and Service Integration

Written by
Aiden Gallagher
and Andy Garratt



A History of Application and Service Integration

By Aiden Gallagher & Andy Garratt

Authors

Aiden Gallagher is an Integration Consultant for the UK and Ireland working with API Connect, ACE, MQ and DataPower® products. He joined IBM in 2015 having graduated from Nottingham Trent University with a degree in Computer Science.

Aiden's areas of expertise include Open Banking, Cloud Deployments and integration. He has written extensively on the history of integration, NodeJS, Agile and API Connect.

You can find a list of his articles, node modules, achievements and projects on his linkedIn; https://www.linkedin.com/in/aiden-gallagher-a5a698b7 and his personal blog; http://aidensgallyvanting.blogspot.com

Andy Garratt is an Offering Manager for IBM Application Integration based in IBM's Hursley Lab in the UK.

Andy has over 20 years' experience Architecting, Designing and Implementing integration solutions using IBM software in the UK, Europe and further afield, ranging from large multi-national enterprises to smaller business and public-sector engagements.

Andy has helped drive the evolution of integration from hub-and-spoke through SOA and ESB, and now onto Agile Integration Architecture utilising iPaaS, Containers and Clouds. Andy enjoys travelling the world meeting customers and partners and finding out about all the great things they're doing with IBM App Connect.

Whilst the authors work at IBM, IBM is not affiliated to any of the content of this book in any way, all opinions expressed within are our own and do not necessarily reflect those of IBM.

Acknowledgements

This book was inspired by the wisdom and patience of my colleagues who sat and discussed the history they experienced, explaining and amending preconceptions of integration and the way in which the computing industry adapted and evolved.

Much of the integration described here was written, conceptualised and advanced before I had even begun secondary school meaning much of that written has been captured from the sentiment and recollection of others.

For my part I have endeavoured to cross check information and validate what I have read and seen with secondary sources of information where possible. I hope to learn from these past encounters to evolve integration approaches and software to continue to improve integration for the benefit of all - Aiden Gallagher

Special thanks to Kim Clark and Dave Hay for their contributions to the review and adaptation.

Summary

Businesses, organisations and governments use applications and computer services to deliver content and business services to their customers and partners. To do this effectively they need to be able to communicate with other departments, employees and enterprises.

This communication and the sharing of information between them is known as application and service integration and has slowly transformed throughout the years alongside technological advances such as the personal computer, mobile devices and the internet.

The following pages take a look at how integration techniques, terminology, methods and products have evolved over time, taking islands of data and turning them into webs of communication.

Specifically, the history of application and service integration will be discussed focusing around the products that do and have existed - typically known as middleware - to enhance solutions or resolve issues with integration.

The book looks specifically at historical events that have led to our current integration capabilities. There is no deep dive into specific functionalities or capabilities of individual products but instead the overall type of product i.e. messaging services.

Contents

Introduction

Integration is the concept of bringing together different and sometimes diverse entities into a single harmonious entity. In computing this means connecting machines, applications and services to bring a smooth experience for the end user.

In this book we will look at how integration has evolved to solve some of the biggest issues when connecting systems together. We will look at how technology has improved, and methodologies have adapted and the resulting benefits and constraints.

Enterprise Integration

Silos of Information

From the arrival of business machines into the workplace with punch timecard machines, typewriters and personal computing, enterprises (government and business organisations) have created systems that deal with specific large-scale tasks such as documenting hours worked, making payments and tracking stock.

As systems, data storage and methods of use have adapted many have remained departmentalised and exclusive, where functionality is or was delivered in a very specific way. These systems were very good at achieving isolated functionality but were often unable to communicate with the wider enterprise to achieve much bigger and more exciting goals.

One example would be in the retail industry, where stock management systems and sales systems were once isolated to individual stored. With the introduction of inventory management, companies were able to start finding and planning for trends across multiple stores using combined sales data. This meant that companies could ramp up stock production and attainment at key times to handle increases in demand, such as turkey purchases over the Christmas period. In a more advanced scenario, integration with external parties like weather companies allows retailers to stock up on umbrellas before a wet season or BBQ ingredients before a hot period in the holidays.

As industries have matured so too has the way in which they integrate; sharing data, connecting applications and systems and exchanging packets of data for the reuse of key business information. This move away from independent systems known as “silos” in the OSIMM Maturity Matrix would become part of a wider technology type known as “Enterprise Integration”, a term coined early in the noughties. [1][2][3]

Some of the earlier integration software solutions, such as; ERP, PeopleSoft, JDEdwards, Siebel and Clarity allowed disparate systems to come together. Where both systems had the

same software solutions they could integrate fairly easily. But, as businesses began to integrate with each other they soon found that these early software solutions struggled to integrate with their peers, requiring an evolution of technology so that integration could become more universal and less reliant on a specific language, hardware or software. [2]

Slow and Unreliable Networks

This aim is constantly hindered by a wide range of issues, of which, the primary concern is almost always the networks that connect each application, service and system and the fact that they always were and still are unreliable. Each transported packet of data relies on either a single or series of networks, where connection can be affected by types of cables, upload and download speed and latency caused by distance from system A to system B.

More disruptively, over long distances bottlenecks become more likely on both Local Area Network (LAN) and the Wireless Area Network (WAN) as the data passes through phone lines, LAN segments, routers, switches, public networks, and satellite links where each cause their own delay which can amount to large wait times. [4]

The biggest issue with sending data is one of physics, in that data cannot travel quicker than the speed of light and whilst satellite data travel is good, the distances needed are huge and as such it is still quicker to send data along a cable having the effect of reducing latency between the connecting systems [5].

Within these Local Area Networks (LANs) where PC, routers and switches are connected, there sits the most widely used network cables - Ethernet. These wired cables offer an internationally adopted standard and also gives additional benefits like prioritisation of sending and receiving data messages as well as an ease of expansion whilst providing control.

Additional systems can be added to the network with relative ease, but the technology is not without its limitations. Cables have to get between the two connecting locations, the cable wires are thin and fragile which can break the whole cable and the most important limitation is the cable length which have a maximum distance of 300 metres, with each metre up to that limit increasing the risk of interference or loss of signal.

Part of the solution - to send data faster - is the widespread adoption of fibre wire connectivity, which is becoming more and more widely used, with huge submarine cables providing under ocean connectivity across continents making networks much more reliable, but still fragile [6].

System Availability

Aside from effectively communicating over a network, there is also the issue of both systems being able to communicate at the same time through system availability. When a

system or the underlying software becomes unavailable it can cause a loss of data, the blocking of processes and lost time for consumers, partners and the enterprise themselves.

Striving for Highly Available (HA) systems - meaning little to no down-time - has been the pressing concern for many enterprises. Hospital life support systems, police communication systems, bank payment systems, railway red/green status all need reliability by being continuously available.

More dramatically, is the need for disaster recovery (DR) a process for when a business site of system operation has been destroyed, damaged or lost all contact with the outside world. Whilst obvious causes; such as natural disaster are few and far between in the UK, there are a host of scenarios from flooding, war, fire or even something as simple as a power surge that can lead to an entire site losing its systems, its runtime and potentially all of its data.

Mitigation efforts normally include a second - and in some cases a third - site where backup of system configuration and data are stored. In the event of a disaster in site 1 the second site is, hopefully, able to seamlessly take over the tasks of site 1 and resume service as normal to the end user.

Sites should usually be close enough so as not to have too great a network disruption such as the network issues mentioned earlier but must be far enough away as to not share dependencies that would cause both to be susceptible to parallel failure. It is also a valid concern for disasters to include malicious attacks, corruption of data and even accidental removal of configuration and data.

To achieve this, enterprise have multiple solution options to keep systems up and running or to recover in the event of a full system failure. Some of these solutions include replication of data, mirrored setup and various backup considerations.

The use of redundant components such as network connections, Storage Area Network (SAN) connections and electrical power inputs mean that outages or disruption of cables, switches or interfaces do not lead to system outages and the components themselves do not become a single point of failure (SPOF).

Loss of data can be protected against at the storage level, by replicating the data across multiple servers both on the same site and across sites for both HA and DR accountability. The data is then preserved in the event of an issue arising on any one server. In order to maintain each server's understanding of the version of truth, servers can use clusters and cluster quorum to allow concurrent reading and writing of data between all the grouped nodes. If one of the servers goes down, the others in the cluster are available with up-to-date data based on continuous synchronisation of the server disk.

Additionally, disk mirroring can be utilised to replicate logical disk across different physical hard disks in real time. The disks can be used to recover to the last replication time and can also be accessed separately for independent reading.

One such solution which utilises disk mirroring is Redundant Arrays of Independent Disks (RAID) - formerly Redundant Arrays of Inexpensive Disks - which utilises five levels to improve on performance, reliability, power consumption and scalability of a Single Large Expensive Disk (SLED). A paper, written by Patterson, Gibson and Katz at the University of California in 1987 discussed these five levels and how by having multiple, cheaper disks an array could have much greater reliability than a SLED. [118]

Some other physical and logical considerations required for enterprise bases around the number of servers that exist for any given application and their dependencies on other applications availability, the level of isolation servers and server clusters have - whether virtualised or not - and the speed for recovery of these servers.

There is also the need for enterprise to plan for and understand the impact of both emergency and planned maintenance on the availability of a server, its applications and its data. Some systems have a requirement for continuous availability which has three main points;

- Ability to withstand single or multiple component failures
- Ability to continue to service customer requests if there is a catastrophic failure
- Ability to introduce scheduled change without disruptions

To satisfy this, enterprise need to be able to deal with cross site and inter site failures whilst still being able to apply fixes and maintenance. For example, when an application level fix is needed across all of the applications servers. [119]

Thought processes surrounding Consistency, Availability and Partitioning have caused much debate within enterprise, with cost implications driving much of decision making. One theorem from the Autumn of 1998 by Eric Brewer looked at these three factors with the published CAP Theorem appearing in 1999. The Theorem has the driving principle that any system could only have at most 2 of the 3 properties, though he later clarified that this position held trade-offs which meant each of the 3 properties could be abided if not satisfied. [120]

A deeper dive into these trade-offs was developed by Daniel J. Abadi for Yale University who described in a 2012 paper “Consistency Tradeoffs in Modern Distributed Database System Design” with an extension of CAP called PACELC. [122]

PACELC not only looks at the possible trade-offs on CAP, but also looks at the misinterpretation by engineers of Distributed Databases (DDBS), such as Cassandra and Riak, opting to reduce their consistency models based on a false assumption that it was required in order to achieve high availability and the “must have” tolerance for network partitions in DDBS. In reality CAP theorem states that a decision must be made to reduce either availability or consistency if a network is partitioned [122].

In raising this misconception, Abadi expands the CAP theorem to also include an ‘else’ trade-off between consistency and latency in the event that CAP can be met, thus PACELC [121].

Without properly planning for HA and DR situations, enterprise leave themselves open to fines, loss of custom and reputational damage as a result of regulatory service requirements or customer and shareholder expectations.

Message Delivery

Even once having achieved HA/DR, there still comes the issues surrounding individual messages not reaching the intended target. The send, receive and reply elements of messaging can either be synchronous or asynchronous where both options may provide the enterprises required level of service.

Synchronous messaging (sync) means to synchronise two things together. In practice it blocks a process from being used until it has completed its operation, whereas asynchronous messaging (async) does not block the process and instead only initiates the operation, allowing computing whilst the message is in transit.

Synchronous messaging can achieve the same parallelism by ‘forking’ a new process for each concurrent operation. Async, on the other hand, requires additional thought to handle the notification of messages never being received or for picking up a messaging operation at a later stage via some form of identifier (id).

Messaging systems can be designed to handle messages not reaching their intended target, with different message requirements employing different strategies. i.e. a message requesting information on where your local banks are, has no real consequence of not being delivered, however a message which holds payment confirmation from one banking system to another needs to be completed once and once only.

There are several levels of assurance that might be used, guaranteed delivery, assured delivery, at most once, exactly once and at least once or any. Whilst these will be discussed in more detail in the problem solutions there are some driving patterns and system designs that are used to achieve these.

- **Fail-Fast**: At any point that a message looks like it will fail in its delivery, the operation will terminate and fail.
- **Fail-Safe:** Where a message failure looks to cause harm to a person or system, the failure will fail in a safe state. i.e. a railway signal will fail to red causing all the trains on the line to stop, as a failing on green would be dangerous.
- **Fail-Stop:** In the event of an exception circumstance such as a crash of a system, a failure causes the operation to stop.

- **Rollback, repeat, retry and backout**: A message is brought back to the last saved process or state, the sequence of events or process is repeated, if that fails the whole transaction is retried for a set number of retries before being backed out ready for later or divergent processing.

Application Divergence

From an application or service viewpoint, the biggest issue is the vast differences between any two applications or service, where both are continually being adapted and changed to either fit in or ignore the latest trends. One area where this has been seen, is the rapid adoption and requirement for software solutions to allow multitude of inputs, outputs, languages, data types, data standards and definitions to cater to the widest possible developer audience.

This can lead to massive amount of time and energy being put into a software type or to allow specific inputs and outputs which do not bring in the audience or market expected.

In order to deal with this issue, it has become increasingly important to minimise the dependencies of applications and services through integration solutions which use a “loose coupling” method which allows integration irrespective of hardware, software and functional components.

Five challenges of Integration

These key issues make up five well-known challenges of integration, which have and continue to be the cornerstone of integration evolution;

- Networks are unreliable
- Networks are slow
- Applications are different
- Application change
- Systems do not share availability

Initially, developers looked to overcome these challenges using four integration approaches;

- File Transfer
- Shared Databases
- Remote Procedure Calls
- Messaging Solutions

These approaches have each grown and developed over the years, embedding themselves into large enterprises across the globe and more recently becoming smaller and smaller bitesize solutions to work for the micro markets they aim to serve. [4]

1. File Transfer

Daily, in business and enterprise, there is a need for employees and departments to share data and information; this might include design documents and reference information. Initially this data sharing might have been as simple as loading punch cards and tape from one machine and transporting to another, in the recent past the preferred method would have been to use a hard drive or memory stick to quickly exchange the data. The issue with these techniques ranged from outdated technology (punch cards) through to the exporting of viruses and malware and the risk of sensitive data being removed from secure locations (external hard drives).

In the modern era, data is often, or is moving to, being stored in the cloud and is accessed over the browser which is a recent variation to isolated servers that are designed, purchased and installed exactly for this reason. But to get that information from machine A to machine B is to use File Transfer, a series of protocols that allow this sharing of information in a controlled and more secure manner.

When two applications want to share a file, they have to first establish a connection before agreeing a location for the file and a filename to be used, and an agreement must be made on what format the file must be, the exact time of the read/write action and which application will need to delete the file once the sharing has been completed.

This is the basis of any given File Transfer, with many solutions being designed and implemented either as protocols to be utilised by applications and services, or in the form of systems capable of performing file transfers.

On 16th April 1971, Massachusetts Institute of Technology Graduate Abhay Bhushan completed the draft submission of a *File Transfer Protocol (FTP)* - a network protocol which utilises *Network Control Protocol* (NCP) to allow bidirectional transfer of files between a server and a client [7]. This meant that systems could now be updated and changed but also pull back all the changes that had been made by others, eventually leading to the synchronisation of files. Enterprise could now communicate with employees from a single server, uploading important documents such as holiday requests or time sheets.

The following year, new addition to the Defence Advanced Research Projects Agency (DARPA) Robert E. Kahn aided first research into a concept of an open architecture of internetworked communication protocols known commonly as *Internet Protocol Suite (IPS)* an improvement on the earlier APRANET which innovated in the virtualisation and layering of networks. It was the additional improvements on NCP which truly made IPS the leading-edge protocol suite.

This suite of Internet Protocols consisted of two new protocols; Transaction Control Protocol (TCP) which was designed for file transferring, and Internet Protocol (IP) which was used for addressing and forwarding simple packets of data.

Whilst originally known as IPS, the suite is known now as TCP/IP, providing application to application specification for how data should be communicated, included how it should be packetized, addressed, transmitted, routed and received [10], [11].

File Transfer is now a cornerstone to many modern businesses. As stated previously, many of these servers are now being hosted on the cloud, either as a larger infrastructure that business can access at will or hosted by third parties. Both of these, however, might be a cause for security concerns despite the ease and speed it gives enterprise.

2. Shared Database

Whilst File Transfer allows the duplication or movement of data from one system to another, it does not satisfy the enterprise need for sharing live data. Often it is important for systems and application to access, append, amend and delete information from a central location, where there is a single version of truth.

To share each individual alteration by file transfer would mean running vast amount of processes to keep every application and file system across a network up to date. Much simpler in terms of data governance, is to retain information on a shared database which can be connected to and updated by applications and services.

One example that explains this well is a shared document that has been put online, either in the cloud or on a business intranet. Without a shared database, every time somebody wishes to edit a file which multiple people use, they must make changes to the document, give it a new version number and then distribute it to everyone it concerns. The issue is that people may be left out the of the forwarding email meaning they are working on old documentation or possibly two people have made changes simultaneously creating version 2 of the document twice.

Underneath the email service, there are file transfer protocols at work, but once you put the document in a shared location such as box, google drive or an internal SharePoint then every person with access to that file (database) can get the current, correct information from the file to amend, delete or create information.

This is where databases come in useful. They are easily accessible by all the systems that require them and keep a current best record of the data they are preserving. Everybody then either queries or alters the database as permitted and needed.

Technically, a database is a searchable computerised system used to store, update, read and delete information [13]. Databases are used to represent data as tables, queries, schemas and reports with the sole intention of being able to process the information in the best way possible.

This became significant to enterprise because it meant lots of data is easily available to different departments and user groups. For example, in an employee management

database, human resources (HR) who wish to pay an employee each month, will need employee bank details, hours worked and their payment amount. Their manager will not need this same data day to day and will instead need to see availability to work and skills summary. Two departments using different elements of the same data set.

Instead of each user having all of the employee's information, they can instead query the database for the specific data they need, making data centralisation and management easier as the size of the returned data is, or can be, reduced.

In order to manage the data being used, there are a number of different Database Management Systems (DBMS) which can be used by applications as shared databases for better integration of systems and applications. A shared database has the advantage of being convenient, quick to develop, easy to understand on multiple levels, and makes synchronising data simple and consistent. However, they are not without their own limitations and drawbacks.

Databases face their own issues ranging from locking of rows and the database as a whole, contention of truth, governance, ownership and latency. In the case of locking, some databases lock rows when in use which makes them unavailable as they are being written to. This then raises the concerns on who has the rights to write to the database, who gets to overrule a change made by another user and what level of consensus is required for changes to be made.

Latency is another issue which can be caused by enterprise centralisation, causing delays in queries to distant locations needing data. As well as this comes the issue of managing the database, making sure the data written is true and that recovery from a backup has a short recovery point objective (RPO).

Additionally, databases themselves are not uniform and come in many different types such as; operational, hierarchical and relational [14]. Below are some of the earlier and best-known software solutions and their implementations in business.

One of the leading Hierarchical model databases is *IBM Information Management System (IMS)* which works by forming a tree-like structure of parent and child nodes, where each child node can itself have its own children to whom it is a parent [16]. Its benefits were immediately obvious; it was a preconfigured standard environment for processing and executing transactions which better allowed business and enterprise to quickly begin new projects that required transaction instead of the laborious task of constantly making their own. This also provided better integration between systems and meant existing IMS resources that were no longer required in one place could quickly be recommissioned elsewhere.

In 1961, Democrat senator John F. Kennedy (JFK) of Massachusetts became the 35th President of the United States with a view of solidifying the USA into history with the first

successful manned mission to Mars, stating: “First, I believe that this nation should commit itself to achieving the goal, before this decade is out, of landing a man on the moon and returning him safely to the earth” [109]. Whilst not achieved until 1969 under the Lyndon B Johnson administration, it symbolised a decade of rapid technological growth.

To fulfil this dream, the National Aeronautics and Space Administration (NASA) began developing, funding and outsourcing its computer programs, pushing evermore for improvements in calculation times, resource management, data processing speed and automation of services.

One such contract, given in 1965 to the company American Rockwell, was to build the Apollo program spaceships which led to a partnership with IBM with the specific purpose of creating an automated system to be used to order and coordinate the materials required to build a spacecraft [15].

Together, they began to design an Information Control System (ICS) and data language interface alongside members of the Caterpillar Tractor company. Working quickly to deliver on the JFK declaration, they had released a first version of the ICS product by 1967. Over the next two years they would add functionality until the first ‘READY’ message was seen on an IBM 2740 typewriter terminal. In 1969 a rebranding to the better-known name of IMS saw it delivered to the IT world, beginning the database management system revolution [15], [16].

The following year, the paper “A relational model of Data for large shared banks” was written by Dr. Edgar F. (“Ted”) Codd as an IBM research project. In the 10-page paper, he detailed an idea for data points held within a table, where each relation consists of a column which holds an information group or category (for example employee name, employee address etc.), and rows which contain data known as entities (for example Joe Bloggs, 1 Acorn Road, etc.).

Each row must have at least one entity which is a unique identifier, known as the primary key (for example employee number, this is not the same for any two employees). Relations can also be made between the rows, and this is known as a foreign key (for example Joe Blogs and Sally Smith both work in the same office) [18].

One primary issue with the systems were that changes to the database could lead to the applications that relied on them breaking. It also meant the day to day users would often find the system had changed underneath them and they needed to adapt the way they worked.

Dr. Codd summarised that “the variety of data representation characteristics which can be changed without logically impairing some application programs is still quite limited”. His idea was to decouple the database, providing independence by separating how the information was stored and how it was presented [106].

The relational database method began to gain support amongst those within the computer science field standing in opposition to the conventional DBMS. In 1974, Dr. Codd and Charles W Bachman stood in debate discussing the two system types. The issue was, that up to this point, there were still no practical examples of relational systems in the industry.

For Bachman's part, he had created the Integrated Data Store (IDS) for General Electric in 1963 which provided a set of powerful commands to manipulate and access shared data from across an enterprise landscape, a working example that was proven to be of value to business [107].

Prior to this development and other localised systems like it, companies had been limited in the amount of business functions they were able to complete using databases, the key examples being small and independent data sets like payroll or stock management.

In labs across the world, scientists were developing systems for relational databases that were to be released in the late 70s and had eclipsed DBMS by the late 1990s. One of the most prolific languages - that still exists today - was the early *Structured Query Language (SQL)* developed by Messrs Donald D. Chamberlin and Raymond F. Boyce of IBM.

This early query language, called Specifying Queries as Relational Expressions (SQUARE) and based on Dr. Codd's campaign for relational databases selected database data using set theory and predicate mathematics. This was eventually refined and expanded to include retrieval and later renamed to SQL due to 'SEQUEL' already being a registered trademark [19], [20].

These systems allowed more intelligent manipulation of the database which was especially useful where access to large banks of shared data that was also accessible across departments was very important. This work would lay the foundations for future relational database management systems (RDBMS).

The eventual release of IBM SQL/DS (1981) and IBM Database 2 (IBM DB2) (1982) were to see exceedingly quick growth in the early years of their release but were so slow in their creation that two competitors were able to develop and market relational databases based on the IBM research before SQL and DB2 could be released.

The first, was by a team at Berkley who had read and began working on a physical representation of Codd's paper which they called INteractive GRaphics and REtrieval System (Ingres), and sold for a modest fee to anyone who would pay for the UNIX operating system database.

At the same time, a former IBM programmer Larry Ellison alongside mathematicians Ed Oates and Bob Miner started Software Development Laboratories which would later become Oracle Systems Corporation in 1982 [21].

More recently, the use of consensus contention data sharing systems - such as the Blockchain - by service chains has reduced some of the negative effects of traditional record keeping that spans multiple enterprise departments and multiple enterprise.

A single ledger is created for the stakeholders in a transaction, whereby each new transaction becomes a 'block' and in order for the block to be accepted into the network each party must give consent to the transaction - which cannot be deleted - but where confidential data is shared only to the parties that require it.

The benefits of these types of systems are that the paper trail created by each party is now in a single immutable ledger where no transaction can be deleted preventing fraud and accidental loss. Additionally, there is a reduction in possible erroneous data being shared since each of the parties must consent to the 'blocks' addition which it can do with trust as each member of the Blockchain 'runs the code' themselves assuring trust in the system and the records held within the ledger.

As with all technologies there are constraint considerations in Blockchain ledgers which are influenced by the primary benefits. For example, a ledger can be compromised if bad agents are able to form a consensus. Additionally, its immutability means that data cannot be deleted, this is a concern with personal data which protected in some cases by governments and public bodies i.e. consumer data rights enforced by the EU through General Data Protection Regulation (GDPR) in 2018 [138].

3. Remote Procedure Calls:

Where the database revolution had dealt in data, it was still tricky for enterprise to quickly repeat and reuse applications and services, especially those that did not share a local address space i.e. a remote desktop. Given the opportunity to do this - and to do it successfully - applications would be able to interact directly with the remote system without having to physically do any procedures, by exposing specific functionality of one application to another. [22]

In real terms the issue was that when system users were getting interactivity with a remote system - in order to perform functions on its system - they would find themselves in an alien world using operating systems they didn't know and trying to call procedures they didn't understand.

Another question which arose was how System A could talk to System B so that when the user had remotely connected to System B, they would be able to use the resources and applications on System A in a System B application or service.

The concept for a solution to these problems was written about by James E. White in the paper "A High-Level Framework for Network-Based Resource Sharing" in December 1975 [23] with a view of moving to an application to application protocol. It was White's opinion

that if applications can perform command-response procedures, then so too could processes via a specific procedure call model.

The new model would allow arbitrary named commands to be invoked by the remote process and output the response both to the invoking program and the invoking user, much in the same way it was achieved before. But additionally - in the new model - single commands would be allowed an arbitrary number of parameters as well as allowing for a variety of types such as integers and character strings and most importantly, to be able to return a usable parameter as a result of a remote procedure.

Some other changes meant that there was now less of a server/client model but instead a peer to peer model allowing procedures and commands to be invoked freely by both systems onto the other and permitting two or more commands to run concurrently [108].

With this new model, enterprises were able to initiate request/reply commands and procedures on remote systems with additional functionality as listed above. For example, with remote procedure calls, a business could now make several remote commands on System B from System A, and when it had both responses it could use them to invoke another call on its own system. All without human interaction.

The model would later become known as Remote Procedure Calls (RPC), Remote Procedure Invocation (RPI) or Remote Method Invocation (RMI) with its first practical implementation in business by Messrs Andrew D. Birrell and Bruce Jay Nelson in the Cedar environment for the company Xerox [24].

Their work, Lupine, was heralded in a citation for the ACM Software System Award received by Birrell in 1994 as having innovated on RPC runtime protocols alongside other procedural methods such as automatically compiled stubs and support for handling exception. Its purpose was for easier distribution of computational tasks that could also be quick but more importantly secure. This differed from the existing implementations which had struggled or ignored entirely the issue of protecting data in transit.

With the success of this program and others just like it, enterprise were now able to securely invoke processes remotely, making enterprise more agile, efficient and safe.

4. Messaging

Whilst RPC's dealt with being able to execute commands on another system, there was still a need to share messages between systems. Typically, one application will want to send messages to a data processing application. It can either write all of these messages somewhere on the expectant system or it can 'queue' the messages for processing at a later time.

An example might be when a payment is made in a local retail chain. The current user of the system might have sales information - such as upselling - recorded for their later

performance review and for later financial evaluation of the company. Whilst the end system which collects all the stores data might constantly be running, it might instead wait to perform actions on the data in a one go during a period of less usage.

It is beneficial for the stores to not have to call the end system, as it may be in use and in effect 'busy' at any given time. Much simpler is to send a message to the system and allow it to deal with each one in its own time.

This integration of applications, which establishes a connectivity method - known as channels - allows the asynchronous sending of - ideally small - packets of data (messages) between the two applications.

Specifically, these messages are published to one channel where they are later subscribed by the receiving application. There are three main types of Messaging that at times overlap in terms of functionality but have key differences in the way in which they work, and more importantly in what circumstances they would be used. These are; Queueing, 'Publish and Subscribe' and the Topic model.

For the integration of the three types to succeed, both applications that are communicating must first agree to establish a channel between them and the format which the messages being sent will take.

Queueing

In Message Queueing, applications make use of an object called a queue, which is allocated space in memory or on a disk to allow the "queueing" of messages.

Messaging and queueing are not a new concept, and dates back in the computer industry as early as the 1960's. Old batch punch card operations were stacked in "queues" with the punch cards themselves holding some form of "message". The messages were pushed onto a card reader before becoming processed using a program library. Some of that data was stored and some was later printed for output. Thus, was born the early concept of messaging [26].

Throughout the 70's and 80's transaction monitor facilities like IMS and CICS became the norm as applications became more and more highly available. Towards the 90's it was realised that some form of queueing was valuable in enterprise and needed to become more available for different network protocols, operating systems and delivery types [26].

There are lots of benefits that queueing messages can bring in terms of integration between systems and applications. The first being that there are no direct connections between programs, and the communication can be separated by time. A system can request a payment be persisted to a database, can continue interacting with the client and receive confirmation at a later time, upon which it can send a confirmation email or message to the client.

Processing of message on queues can be driven by either a message arriving on a queue which trigger an application, or by events such as a given number of messages now waiting processing or certain priority messages needing to be processed.

In the beginning of the MQ story there were few programs available with a market that was fairly cornered. The two main benefactors being IBM MQ and Microsoft MQ.

IBM Message Queue (MQ) Series - IBM began to utilise messaging as part of their integration approach in early 1992. By 1993 they had brought proprietary for ezBRIDGE (an IBM MQ precursor) announcing their intent to release the early MQ Series. In 1994 they had acquired ezBRIDGE and released the first of the messaging software solutions.

Microsoft MQ – Microsoft first released their message queueing technology in May 1997, creating what is now a somewhat common Windows Operating System MQ platform.

SonicMQ - On October 4, 1999, Progress Software - a global software that deals with integration and data solutions - created one of the first messaging middleware with Java Messaging Service (JMS) capabilities. Additionally, upon its release Sonic has begun to utilise graphical interface and XML into their design gaining them a Java Developer Journal award in 2000 [124] [125].

Publish and Subscribe

Publish and subscribe models, refers to the model whereby one application waits for a certain event before publishing to its listening (subscribing) community. One example of this in day to day business would be waiting for a user to have completed a service free trial period, such as free Wi-Fi or membership to online streaming services.

Once the event occurs, the publisher can publish the information to a channel, or channel topic. The subscribers will be constantly listening or requesting any new data added to the channel. Once new data appears, it can be processed and then an action derived from it.

An example of publish subscribe in action would be a football club scoring a goal in a local game. The football club or observation body would publish the goal along with some data including the time it was scored, who scored the goal and the method for scoring. Lots of applications and entities use that data for their own interaction with their consumers, for example a betting company will want accurate information in order to pay-out, invalidate or declare a betting slip as void. A sports news centre would want to announce the goal being scored either indirectly or through a push notification service, the football clubs themselves would want a record for their contractual agreements such as goal bonus awards.

The publish and subscribe model allows this easy sharing of data between those wanting to push and those wanting to pull information. The benefits of this model are the ways in which it allows decoupling, by not replying on both the sender and receiver both actively

participating in the exchange at the same time. Time decoupling allows one of the entities to be offline and still be able to handle the exchange when they regain connectivity.

Access Control is also an area which benefits from decoupling within this model, space decoupling is accessible when publishers and subscribers interact using the event server, data could be driven by multiple unknown entities that are involved in the data transformation prior to publishing, additionally, the subscribers could also be unknown to the publishers themselves.

Finally, events themselves can be decoupled along with the processes that handle them through synchronisation decoupling. Subscribers can be processing an event and still receive notifications of other events occurring, whilst publishers are not blocked when producing events [148].

Whilst this level of decoupling is an advantage, it brings with it an element of disadvantage in that there can no longer be an assured delivery method simply because the publisher and subscriber no longer have a direct line of communication. Instead tight coupling would be required at another stage in the flow such as the subscriber publishing read receipts.

There also lies a problem in scalability, where increased message flow, subscription and publishing can cause the system to slow or max out the broker being used by a load surge at a given time. A good example would be when an incident happens in a football match such as a player receiving a card. This would create a surge as all the subscribers begin receiving messages. Scalability also depends on the existing architecture that it is using, which if not correctly sized can exacerbate slowdowns and load surges.

Topics

Instead of using the queue model, it is possible, and in some cases advantageous to instead create topics. Using the topic model, a server can expose a subject, where publishers can write messages to and subscribers can read messages from.

The topic method allows multiple subscribers to find out about messages on a topic without individual connection being made on a one-to-one basis between all the active entities. Instead they connect using a broker allowing a many-to-many relationship for both publishers and subscribers of messages.

One example could include a taxi service which allows mobile device users to request a car to pick them up and drop them off. In this system there are taxi drivers and there are fare payers and all users must have access to the taxi application. The application company, i.e. the journey broker creates a topic:

/taxi/journey/{city}/request

Each new fare payer who requests a journey would have their location inserted into the {city} field and a message published to the journey request topic. At this point, the taxi

drivers are subscribed to the same topic with their own location mapping determining the city to be subscribed to.

As each new fare payer requests a journey, all the nearby drivers get the request and can respond. Whilst topics are available in most MQ software, the full queue package may not be necessary and can add cost to an enterprise operation.

More recently the benefits of a lightweight MQ has been understood and accepted by what was originally hobby programmers and system builders. With the growing ease of sensor activated event-based messaging to initiate other programs, has come the need for a smaller, easier to implement message queueing technique.

Unlike the full MQ systems, Lightweight MQ works using only the publish and subscribe concept that relates to MQ topics. Where real time data is being gathered through sensors, meters, actuators etc, some machines need to speak to other machines (M2M) to trigger events or processes. For example, in Marwell Zoo, MQTT (MQ Telemetry Transport) is now being used by presence tracking sensors that turn on heating for the animals when they are in their habitats. In another application, live GPS signals on ferries can deliver real-time data via twitter to give precise arrival times to passengers.

These sensors and meters which communicate using a central broker, publish messages to a topic on the broker server. Relevant meters, heaters, motors etc. which are subscribed to the topic, and then get events in real-time to initiate a task.

Apache Kafka - Whilst Apache Kafka describes itself as a distributed streaming platform it is driven by the publish and subscribe capabilities of messaging. It uses these capabilities to allow streams of records to be produced (published), consumed (subscribed to), streams (server) and Connectors which allow connection to other integration products such as a database [129].

It was around 2010 when a team at LinkedIn first began to develop what was to become Apache Kafka. The company was, at the time, beginning to move away from traditional monolithic messaging types and adopting microservices. From that came the need to constantly process data that was streaming from the user interacting with the platform each day whilst maintaining message consistency throughout. It soon became successful and was released as open source in 2011 [130].

The Cloud has been seen as a huge benefit for IoT systems - which clients of Kafka are likely to be creating - by providing the ability to quickly scale up or down when sensors are active or busy as new virtual machines or lightweight server are created on demand.

At times it has been difficult for various enterprise to get the different Messaging platforms to be able to speak and inter-operate, an irony not lost on the integration community. In 2003, John O'Hara of JP Morgan Chase determined that in some scenarios such as in banks and stock traders, there was an economic impact when messages are lost, late or corrupted,

becoming especially true for the high volume, high specification environments that messaging middleware was being deployed to.

Lots of the enterprises that fit the above description relied upon existing technology, but since many could not interoperate easily, the gaps had to be bridged and internal solutions found which were costly in both time, energy and money. Instead, O'Hara sought to bring together Messaging solution companies in order to create a protocol to be used that was collaborative, independent and free from royalties [126].

Advanced Message Queueing Protocol (AMQP) was soon adopted by a large host of enterprise such as; Microsoft, Goldman Sachs, Apache, RedHat and many more. The basis for the protocol being formed into 'frames' to enable the transfer of messages between two solutions. Each frame body allows one of nine 'performatives'; close, end, detach, disposition, flow, transfer, open, begin, attach and transfer. [127]

The protocol describes 'distribution nodes' which acts as a communication point between senders and receivers to allow a mechanism for move and copy, the ability to create nodes as and when they are needed and refine messages through filters [128]. Finally, it also defines an immutable 'bare message' which is to be sent from the sending application through each process. The message format includes a header which details the 'time-to-live' and priority with the bare message itself able to detail a host of optional variables such as a correlation ID, message ID, reply to, and user ID.

Where Messaging excels, is in providing methods for assured delivery by providing a level of guarantee that messages will be successfully delivered over an unreliable infrastructure or network. This has real benefits for enterprise methods which require 'once only' or 'at least once' delivery of key messages such as payments from a user account.

To achieve assured 'reliable message' delivery, messages are stored in the transmitting applications memory whilst the subscribing applications must give acknowledgement of having received the message. If the message is confirmed as received and written to disk it is released from memory.

When a subscribing application returns from not being available, it will give a recovery point or unique subscriber key to a 'coordination manager' which is configured to listen on the same data channels as the receiving applications. If the coordination manager is unable to store the messages quick enough, the transmitting application will block until the current messages are written on disk [137].

By using asynchronous messaging, publishing applications can send information to a dormant or waiting application without blocking their own processes. One example where assured delivery may come in useful is a production reporting system which waits for the full production data for the day. Throughout the day, different systems can send their data to the reporting system which begins processing out of hours, all the messages are assured

delivery, but each system has not been blocked waiting for a response from the reporting system.

Middleware

These solutions; File Transfer, Remote Procedure Calls, Shared Databases and Message Queueing were designed to solve the problems around integration and have succeeded despite and in harmony with each other. Their evolution has come over decades, reusing old technology in new ways usually to solve small issues which expands and develops over time to become the vast software and hardware solutions that enterprise use and the world touches every day.

This family of solutions known by the umbrella term of **middleware** - coined back in 1972 - describes the mediation of software resource(s) and a software application. In 2003, Nick Gall described middleware as being “software that mediates software”. As described in this chapter, middleware software has looked to bridge the gap between disparate systems and bring them together into easily accessible, reusable and manageable resource for enterprise to deliver to their customers [27].

Middleware in itself comprises of a wide range of technologies through messaging solutions, streaming of data, portals that combine multiple interfaces into a single easy to use site. Solutions will often comprise of adapters and connectors to ensure ease of integration which brings each of the data points together. Further details on these solutions and products will be discussed throughout with the benefits and limitations each brings.

Everyday Integration

In this section there will be a brief look into some of the existing solutions that exist today with examples of how it might be used day to day to those both in and out of the integration world.

eMail

Electronic Messaging (eMail) has been around in some form since 1965 after MIT had begun to link up academics using the IBM mainframe system MAC IBM 7094s and the underlying Compatible Time-Sharing System (CTSS) in order to ‘share’ messages in a shared directory. At the time messages would be stored in directories like ‘To Tom’ which would later be ‘RETRIEVED’ by the recipient. Later a user would have their mail placed into a directory called ‘mail box’ in their home directory where other users could only append to the directory, thus securing it for the owner.

Other facilities that could have been using eMail at the time, in some form or other were the military, SAGE, Honeywell and IBM at which time MIT were using Multiplex Information and Computing Service (Multics) - another time-sharing mainframe operating system. It was

around 1972 when researchers at MIT such as Tom Van Vlecks who were developing their mail system were approached by J.C.R.Licklider of APRANET with a view of connecting the various APRA-funded machines via the internet to deliver mail.

The Multics system was connected to the APRAnet in 1971 before a networking group led by Mike Padlipsky - a then member of the Multics development team - and a member of Project MAC - a DARPA funded research laboratory. At the meeting the team collaborated and discussed sending mail by FTP in which one member described having sent messages previously using '@' between the user id and the hostname of the machine being sent to - contentious only because it was the line delete character of Multics at the time.

Later the same year a small group of developers worked on yet another time-sharing system called TENEX where a test email was sent from one computer to another using solely the APRANET network marking the sender, Ray Tomlinson, as the first sender of an eMail from computer to computer over a network [131], [132], [133], [134], [135], [136].

This use of eMail (Email) over file transfer protocol has since been a major asset for business and enterprise with a staggering 10,000 emails sent per worker per year in 2013 [137] from enterprise wide announcements, sharing of content, tracking work orders, subscriptions to information, meeting planning and general day to day notices between peers.

Media Downloads

Another integration solution is the use of file transfer in order to download media content such as; music, films, books and documents like this one. The movement away from hard disks of file content - which still involves the same middleware solutions of file transfer - onto digital movement of data has made media content more accessible than ever before.

One of the first steps to making media file transfer widely popular both in home and in enterprise was the ability to format internet messages such that the message header is able to give lots of information, but the content of the body is flat ASCII text. In 1992, Nathaniel Borenstein and Ned Freed published Multipurpose Internet Mail Extensions (MIME) in a Request for Comments (RFC) that described how to allow support of multiple attachments, rich format text, unlimited message length and executable attachments such as; video and audio [160].

Further progression by the network group saw advancements made to MIME for the sending of media primarily over SMTP and email but also paving the way for HTTP delivery of media content. With the prominence of the world wide web (www) and the ability to copy files from one server or computer to another - known as downloading - new markets quickly grew moving away from physical files to digital.

A common problem with media downloads was the size of the files being transferred. In order to make download quick and feasible on then slower network speeds, there was a

drive to reduce the total size through compression; a method which replaces repeated snippets of code with reference markers allowing the file to be smaller.

Before: The teacher's tea was the farthest from the teacher's team room
After: $1 $2acher's $2 was $1 far$1st from $1 $2cher's $2m room

In the above example compression reduces 'the' and 'tea' to turn the sentence from 63 characters to 56 characters which can be applied at a much larger scale.

The International Organisation of Standardisation (ISO) and the International Electrotechnical Commission (IEC) created between 1988 and 1992 for standardisation of lossy - irreversible - compression from an analogue format such as Video Home System (VHS) to a Digital and smaller file version. Known as Moving Picture Experts Group Phase 1 (MPEG-1) being applicable for both video and audio, an audio specific implementation MPEG-1 Audio Layer III (MP3) became widely popular amongst music downloaders.

Record companies across the world began capitalising on the digital marketplace - having struggled to immediately recognise the opportunity it brought (something symbolised in the title 'record label') [162]. This saw increases in revenues from 2004 through to 2017 with digital only exceeding physical in 2015 in the US [161].

That isn't to say that downloads were not more prevalent prior to 2004, simply that it didn't translate into direct sales for record companies and other media providers. With the new ease of access to coded format media and the internet - illegal sharing and pirate copying of files through point to point File Transfer became exponentially a problem to the video, software and music industries.

Rapid increase in pirating was possible through torrenting - a method whereby downloads occur across a network of machines which have small 'packets' of the required data made available by a 'tracker'. There is still a single 'seeder' machine which has the file in its entirety, but file download is spread across the 'leecher's', each of whom are also uploading the file for other leecher's.

Torrenting approval is best described through Napster, a software that was created by Shawn Fanning in 1999 - named after his high school nickname - in order to better enable file sharing amongst his peers. In his three core principles he wished to allow;

1. The ability to search for files from a collective database
2. The ability to share files directly and not on a server
3. The ability to talk to other online user in real time chat

The platform achieved precisely what it intended but did not seek to enforce copyright law in place for the multiple record labels whose artists music was being shared. After a request to cease Napster activity by the Recording Industry Association of America (RIAA) in 1999 was ignored, a lawsuit was filed leading to the platforms court ordered shutdown in 2001

whereby Mr Fanning declared bankruptcy despite having been on the cover of Times magazine [169].

Whilst Napster dealt with one to one interaction across a pool of people - alike to similarly shutdown sites; Limewire, Pirate Bay and Kazza - future sites would begin to perform torrenting in its prescribed way both within and outside of the law [164].

In 2001, BitTorrent creator Bram Cohen developed a Peer to Peer (P2P) protocol and platform which saw heights of 405 million 'torrents' in just the first 6 months of 2012 and in 2006 accounted for up to 70% of all internet traffic [163]. This prominence remained over the next five years but saw dramatic reduction in traffic accountability best displayed by the rapid reduction in North America estimates; 18% in 2011, 7-8% in 2013 and just 3% in 2015 [165].

Partially, this was caused by systematic clampdowns on torrent sites [166], their owners [167] and torrent site users themselves [168]. Mostly, the ability for users to legally access the media content - video and music - they wanted became easier, cheaper and without the risk of criminal repercussions.

Websites soon emerged that allowed video content to be watched online through streaming, the most notable being the emergence of YouTube in 2005 by PayPal employees Jawed Karim, Steve Chen and Chad Hurley which allowed users to post home videos online where they were stored and could later be watched - for free - by any of the site's visitors [170]. Whereas downloading pulls files from one server to another, streaming does not require the downloader to retain a copy of the file locally.

Streaming allows the consumption of videos and audio as soon as the data begins to arrive on the viewers device rather than waiting the full time for a download. Built upon three simple building blocks, data files are split into blocks of data, encoded using compression standards and transported continuously as part of a stream.

The encoding of video files which includes its compression was to allow data to be transferred as quickly as possible whilst being able to be ordered by the streaming system. The progression of the MPEG4 standard helped to facilitate the streaming process and in what is now known as either H.246 or AVC which allowed good quality video at lower bit rates with ever improving service between its first publication of Edition 1 in 2003 through to Edition 12 in 2017 [171].

The encoding of audio files was built upon MPEG (MP3) with the publication of the Advanced Audio Coding (AAC) standard in 1997 to achieve better quality sound at the same bit rate by working on the coding efficiency, coding accuracy and allowing more sample rates and compression modules [172].

Once compression is completed, data is inserted into a data container (wrapper) ready for transport which describes the structure of the file, the metadata and index information.

There is a selection of common container formats, proprietary formats such as Adobe Flash Video (FLV), Windows Waveform Audio File Format (WAV) and Apples Audio Interchange File Format (AIFF) or QuickTime File Format are only produced and can be limited for complete playback to associated software's. Other container formats are freely available for production and use such as Matroska (MKV), Real Media Variable Bitrate (RMVB) and Audio Video Interleave (AVI) [173].

Transportation of the stream data relies heavily on the TCP Protocol with early streaming protocols using User Data Protocol (UDP) which was designed in 1980 by J. Postel to provide a minimum set of mechanisms such as removing error handling, at the cost of not being able to guarantee delivery or provide duplication protection [174].

This minimisation allows a greater speed in connectivity when using the protocol, which makes it a good base for streaming which allows fast transportation of data packets. It was on top of UDP that the Network Working Group created the real-time transport protocol (RTP) to facilitate real time transfer of audio, video and simulation data across multiple streams where video and audio can be sent over separate channels to be combined by the receiving program [175].

It is not unusual for new transport protocols to be built based on the RTP principles which was designed to be incomplete to allow customisation for each application that should wish to make use of the specification. Generally, data container developers have built proprietary transport mechanisms for the applications streams arrive on such as Adobe HTTP Dynamic Streaming (HDS) or Apples HTTP Live Streaming (HLS) which forms part of the HTTP protocols at support adaptive bitrates.

Worth noting is the Real Time Messaging Protocol (RTMP) which was created by Macromedia for Flash programs - a highly popular implementation of video and audio on websites such as in the use of online games that rose to prominence by working with most media formats and with variations for tunnelling and different levels of encryption [176].

Streaming as an alternative to owning media content has become the norm with streaming facilities linked to user accounts much easier to manage, cheaper to consume, legal and most likely to have the content desired due to a low barrier to entry.

YouTube, for example, allows social media influencers to get their voices heard much better than any preceding blog technology, music videos find themselves with hundreds of millions of 'hits'.

Other technologies also maximise on the consumption without ownership model with Netflix (TV and Film streaming service) averaging 140 million hours of TV and Film viewing per day in 2017 with on-demand viewing of broadcast TV in the UK having doubled to 8% since 2013 and Spotify (music streaming service) streaming 20 billion hours of music in 2015 [177], [178], [179].

The dominance of streaming is becoming more apparent with record label figures for 2017 showing record sales to be distributed fairly evenly at $5.2 billion of physical sales, $2.8 billion of digital sales and $6.6 billion in streaming sales with streaming up from $2.8 billion in 2015 and $4.7 billion in 2016 [161].

The download of media has moved progressively over the last three decades towards a mass consumption model where ease and speed of access is the predominant selling point for the integration technology employed to reach an audience.

Automated Teller Machine (ATM)

When British inventor John Shepherd-Barron arrived at his local bank a minute too late to withdraw money he determined that a better way of retrieving money from an account must be possible. Whilst considering chocolate dispensers he wondered why the same didn't exist for money. In 1965 he approached the then Chief General Manager of Barclays with the proposal of an automated teller machine (ATM) who liked the concept and installed the first into their Enfield branch in north London on 27th June 1967 [180], [181].

An ATM can now be used for a number of different bank transactions; money transfers, deposit (cheque and cash), top up a mobile phone, pay bills, check balances and change existing PINs for chip & PIN cards and is a popular method with over 50 million transactions per week.

ATMs rely on middleware to process and complete withdrawals. The person using the cash machine first must insert their card into the ATM where a check on the validity of the card is completed against the electronic fund transfer (EFT) financial service serviced by the ATM - for example, Visa which will service a multitude of banks.

In the UK, once the card is confirmed valid the ATM will request the corresponding PIN from the Chip & PIN card which the user inserts using the ATMs output devices. The PIN is sent to the service providers (e.g. visa) ATM server to check that it matches the card. This then issues a validation, rejection or – in some cases - a withholding of the card itself if too many rejected calls are made.

The card holder is then able to access the various facilities of the ATM. Using a cash withdrawal as an example, the user will interact with the ATM interface by selecting withdrawal and an amount. Once confirmed, the ATM will forward the request to the service provider who then forwards the request to the associated bank adding additional information about the card holder and themselves to allow the bank to confirm who is making the request and that they have the authority to do so.

In the bank systems itself the request will arrive as a confirmation of funds against a specific bank account. This might be processed by a number of integration products, which will

provide positive or negative feedback on the amount requested. When positive the service provider will release the funds by authorising cash to be dispensed from the ATM.

Once the cash is dispensed a receipt of withdrawal is sent to the bank by the service provider to allow for later reconciliation. At the bank level these receipts are likely to be stored on a messaging queue to be transacted at a set time in some form of batch process that will occur during quiet hours.

One of the more prevalent methods of storing these receipts is by using messaging integration products. In 2018 alone, more than 90 percent of the top 100 banks used MQ to securely and reliably send and receive messages. Message Queueing products are seen as advantageous as they allow for methods such as ‘once and once only’ transactions, reducing the risk of withdrawals being twice receipted from the account of the ATM user [182].

Whilst these middleware examples give a high-level overview of some of the more common integration usages, the next few chapters will delve deeper into some of the patterns and methodologies used to implement them. This will touch upon data collection, business processes and the evolution of strategies. It will go on to describe how these strategies were expanded and improved from one iteration to the next to create an ever more flexible and agile approach to building applications and services.

Early Integration Patterns

A pattern - a regular and intelligible sequence of events and process - allows previously tried, tested and standardised methods for implementing integration to be repeated rather than reinventing the best approach every time.

Patterns can be kept secret and sold, or shared and improved by the industry and can be adapted in small ways to fit the individual needs of each enterprise. What makes patterns valuable is the ease in which patterns can be reused, the simplification it brings to the organisational structure and the assurance that the method has been tried and tested.

Integration patterns may come in many shapes and sizes, they might describe many-to-many integration or one-to-many integration, they might cover standards for naming and sizes or focus on a specific task such as sending a message or, could look at a full business model including; transport, security and orchestration of messages.

Integration Levels

Enterprise integration is achieved through four specific approaches at four different IT levels; Data Level, Application Interface Level, Method Level and User Interface Level. Each of the levels can apply to any and all business and offer potential improvements from the back-end systems that hold the data, through to the colleague experience of using various technologies and the end customer experience. What the levels offer to enterprise is the opportunity to adapt to customer and internal requirements in order to grow business and improve their services.

This might be a regional manager of a local supermarket aggregating data from different branches through a portal, budget information being integrated with each department's internal systems or an aggregation of incoming news for publishers to scrutinise before releasing their own content.

1. Data Level:

Enterprise look to share data at the database level through the sharing of data, also known as data-orientated integration. What enterprise requires is an approach that allows data to be altered, amended and enhanced between their different systems for ease of reuse.

By using middleware tools to allow for data transfer in both real time and non-real time, data can be moved from one database or datastore to another, being transformed to adhere to a specific format compatible for the end datastore or the communication layers preferred format. Throughout, business logic can be applied during transportation to meet the specific needs of whatever system wishes to use the data.

In a car sale, the dealership creates a contract with the purchaser before completing an

online web or application form. When the dealer commits the transaction i.e. after payment, the data is sent to head office where the required sale information is extracted, the order also goes to a manufacturer where the car specification is input into the production chain, business logic will check stock levels and create orders where there are shortages and finally an estimate for completion will be generated based on previous statistics and a delivery process flow implemented to get the car to the new owner. All using the initial data from the ownership [35].

Data Integration methods historically included batch transfer, data union, data replication and Extract, Transform and Load (ETL) solutions [36]. ETL itself was used from the 1970's as a technique to move and transform data from 'disparate' sources to a target location [37]. Several databases could and still can be consolidated into a data warehouse, with ETL used for the migration of data from one database to another. An enterprise wishing to centralise their membership data from multiple stores into a single database warehouse might use ETL.

Another type of middleware known as database-orientated middleware (DOM) is also used as a data integration method using standards such as Open Database Connectivity (ODBC) and Java Database Connectivity (JDBC).

ODBC was originally developed by Microsoft in the 1990's as a C language programming interface that worked independently of the operating system or database. It's a well-known portable language that allows for great interoperability and ease-of-data access, but Microsoft soon became SQL focused and MySQL has become a modern competitor to ODBC [38].

JDBC was created in 1996 as a standard library for updating and querying relational databases, it was released as part of the Java Development Kit v1.1 in February of 1997. The application programming interface (API) was created using Java by Sun Microsystems [39] [40] .

Advantages: The database-orientated approach allows data to be reused across applications with minimal changes being made to the application code. This makes it a cheap option as less coding, testing and deployment is required, and the DOM technology is relatively cheap [35], [41].

Disadvantages: A common problem with this approach is that business logic is only in the primary application and relies on a data model. In the event of a data model changing, the integration solution may no longer be valid, leading to enterprise resolution costs [36], [41].

2. Application Interface Level:

The Application Interface Level approach integrates the interface of each application to work with enterprises business processes or to gather important information (though usually just simple bits of information). The applications can be custom built or used out of

the box with key developer services exposed.

This involves taking information from “Application A” using the application interface, transforming the information into the intended format, before sending it to “Application B” which can use the information as it likes.

An example of this would be to take patient monitor information from a manual check input into a reporting device or directly from a connecting health monitoring device. Each device may use a different underlying technology - due to age and contracts for new devices - and might each use different formats. At this level all the data points will be collected and formatted ready for use by a central computer used by a hospital matron.

The most common method of this approach is a “message broker” which works by use of a bus or hub to control flow of information and perform transformation on the information during the flow [42]. Message Brokers may also route information across destinations, deal with external databases, respond to errors and mishandled data as well as invoking web services.

A message broker ‘bus’ is named so because it has a flow with a start and finish and at each node it will either transform data, offload data to another source or intake new information - much like a bus takes a route, picking up and dropping off passengers at their required destination.

One of the early message broker products was Neo-Net, a software developed by New Era of Networks Inc. (NEON) with a lot of modern integration bus features offered in v2.2 as a “multi-purpose messaging software.” An initial use for the software was in hospitals with the first being an implementation in the USA hospital of Mulhenberg, on SunOS and Windows NT with a price of $18,000 - $85,000 in July 1996. Hospitals were quickly able to use the broker to centralise their data, implement rules and interface between disparate internal systems [43].

Neo-Net was later used in IBM MQ Series Integrator (MQSI), released in June 1999 after it was rebranded during its acquisition by IBM from NEON. The technology was made into the Rules and Formatter components which were licensed separately from the main software.

IBM rebranded frequently from WebSphere Business Integration Message Broker (WBIMB) in 2003 after a move to eclipse, to WebSphere Message Broker (WMB) arriving in the following release, the latest of these rebranding’s in 2013 saw IBM Integration Bus (IIB) being released after a consolidation of the IBM integration family [44]. In 2018, IIB was further consolidated and is now IBM App Connect Enterprise (ACE) the enterprise edition of IBMs application connection software.

Competitors were slow in the making, even as late as 2002 - 6 years after Neo-Net was released, Sybase, the company which had acquired NEON went on to release their own Message Broker under Financial Fusion - one of their subsidiaries - as part of the TradeForce

Suite [45]. Little else can be found of the particular software suite but it had taken up to the early 2010s until lots of competitors could be found in the message broker space.

Microsoft Azure Service Bus - Originally .Net (Internet) Service Bus Windows released their Cloud Service Bus in 2009 focusing on the benefits the Cloud can give by lowering entry level knowledge requirements whilst also offering scalable and available service bus.

The software allowed for a centralised cloud event hub for publishing and subscribing which had, critically, a relay service in order to reduce the overhead of self-managed internet connectivity such as firewalls (software and hardware), load balancers and DNS setup. Windows were pushing to create their own Cloud system that could - when understood and made easy to replicate - dramatically cut the amount of time to create a new service bus [140], [141].

Fuse Message Broker - In June 2012, RedHat the open source software provider, acquired FuseSource from Progress Software Corporation. It had just the year before been accredited as being a leader in a Forrester Wave report for giving standard-based services that were open source and supported [142].

Advantages: The advantage of an application interface level approach is its ease of application, with lots of interface possibilities for popular applications or services. There is also no need to create middle integration interfaces with technology fast growing to allow expedient adoption of the methods described.

Disadvantages: The downside of these methods is that the software is often proprietary, with few open source and open use technology. There is also a huge disparity between software solutions and for a long time there was not a lot of message broker competition.

3. Component (Process) Level:

The sharing of business logic amongst multiple applications is possible at the component-level approach of EAI, by making use of an application server or method warehousing where business logic processes are centralised, thus allowing them to be reused.

This is important because it allows a single place for the exposure of business logic much in the same way services are exposed for the application interface level. This centralisation reduces the overhead of implementing the same business logic across different departments.

Given an enterprise wishes to make discounts on products or a service based on customer loyalty, rather than each department having the same logic to implement this, a central server provides the logic.

These servers, sometimes known as application servers, are software that reside on a server in the network and holds business logic interacting with one or more clients to provide

business logic for applications when the application flow requires it.

Method level integration is a deeper type of application interface integration used to consolidate existing business methods into a single reusable component. As with the application interface level the use of web services to achieve this consolidation only each of the applications that do so must support RPC or RMI in order to allow business methods and functions to be called. This has its own disadvantages in that each application becomes tightly coupled to the main method application being unreachable when the application is down or causing widespread enterprise downtime when its hardware or software is faulted.

By 1997 the open-source Apache Web Server had become a leading choice for web developers. At the same time IBM were looking to build on their ever-increasing integration platform and so leveraged the Apache software in order to build IBM WebSphere Application Server (WAS).

By building on top of the non-proprietary Apache Software the team from North Carolina, led by Chris Wicher and Sue Wallenborn, decided that adding the extra functionality of transaction processing and message brokering would expand the scope and use of the existing WAS product and in doing so created the highly successful IBM WebSphere brand [54].

Whilst it is possible to code this form of integration on a case by case basis, a common approach for implementation is to use Web Services to share methods [42]. Alternatively, it is possible to share existing methods through distributed objects; a concept that stems from distributed systems and RPCs.

Distributed objects are objects that can be invoked by a remote process on a network connected device. ‘Objects’ themselves entered the computing space in the 1970’s as a way of dealing with software complexity by having the ability to be deployed across a wide range of systems and software specifically by using a ‘skeleton’ and a ‘stub’ to give the impression that a business method is running on the remote machine invoking the process rather than on the server itself.

The early object orientated architecture had numerous and ranging implementations which included; Burroughs 5000, Intel 432 and IBM System/38 which continued to expand with artificial intelligence, database model improvements and advances in cognitive sciences [46].

The original use of object-based development and classes first appeared in the language “Simula 67” and by the 70s languages such as Alphard, Mesa and Modula were developed around data abstraction. Further advancements were later made around abstraction layers by developers such as Liskov, Zilles, Guttag and Shaw [46]. More recently the use of object-orientated concepts within the C programming language in the form of C++ has become a popular base, becoming instrumental in the teaching of coding in universities in the early

2010's whilst C# was being taught in some secondary schools in 2017 [47].

Advantages: There is a huge resource of software available for the method level approach of EAI with new languages being written all the time. This is a field with lots of approaches and is extremely effective when done well with the possibilities for capabilities being comprehensive.

Disadvantages: The issue that can be caused at this level of integration is the modification required on existing applications in order to expose and reuse existing business processes, whilst costs can be expensive as a high-risk method which requires skilled workers.

4. User Interface Level:

Integration at the user interface level is often known as "refacing" and is usually a browser-based solution of integrating PC graphics or terminal screens. It stands as an easier method of integrating interfaces then the "Swivel-chair" approach - whereby a human user has two browser screens open and either copies and pastes data from one to the other or manually inserts the data from one to the other.

Swivel-chair integration might also use hardware storage device to quickly transfer data from one stand-alone computer to another unconnected machine or server by the use of a floppy disk or a hard drive that is physically plugged into the system. But with the introduction of the cloud and shared network prevalence it is more often easier to share documents via an online shared storage such as Box or Google Drive.

Whilst the approach is simple and has a low technical threshold to achieve, it has, and continues to be a suitable option for users wishing to share data inside documents or browsers very quickly. But manual data procedures are prone to human-error, data can be mistyped, miscopied or replicated to an unsecure location whilst the physical storage devices may contain malicious software that can infect machines and servers. A further risk is the copying of customer personal information which can be used by a rogue agent within an enterprise.

Instead of this human approach, enterprise can make use of refacing which can be achieved by bundling the user interfaces of two user systems or browsers, a process known often as screen scraping which can also be used for content aggregation. The benefits to this over the manual approach is the speed in which information can be moved although the setup might take longer. A benefit is that the reliability of the data being the same is much higher between one and the other by avoiding human error.

Screen scraping has a long history though it is often now used as a term for web data abstraction i.e. to take a whole browser page and cut out the bits of information needed, but its original use was on a more local level, specifically on mainframe applications that could only be accessed via the terminal screen. The easiest way to access the data held on the mainframe was to capture the content on screen and transfer it to another system.

Where screen scraping has excelled and been used to formulate new markets and enterprise is its use for content aggregation, which is the consolidation of information from multiple sources which may also use web services and APIs.

Aggregators in themselves can come in very useful, to analyse user experience and sentiment in relation to specific content or services. By aggregating data, enterprise has the ability to formulate business opinions and logic which can be used to offer customer new products, discounts of simply to improve on existing offers.

Some types of aggregators include; Poll, Review, Search, Social Network, video and news aggregators.

Poll Aggregators – These work by collecting poll results and averaging them to give an estimation of who will win in an election. The first of these was used within the website RealClearPolitics by Tom Bevan and John McIntyre who founded the website in the year 2000 from their Chicago apartment [48].

Review Aggregators – Companies like Rotten Tomatoes, IMDb and goodreads use visitor data to review products such as; cars, films, electronics, video games etc. The earliest review collation was founded in 1993 when Stewart M. Clamen founded the Movie Review Query Engine [49].

Search Aggregators - These systems use results and data from multiple search engines providing a wider range of control over content. The idea was originally published by Apple in 2005 after a patent file in 2000 [50].

Social Network Aggregators – This works by collecting multiple user social network information into a single location. It is often used by companies to expand and manage their brands and can also be used as a source of financial news to capitalise in the markets [51].

Video Aggregators – A video aggregator is used to collate videos and then expose the videos based on user preference, previous views or most viewed in terms of a wider audience. This can be seen in companies such as YouTube.

News Aggregators - Also known as a Rich Site Summary (RSS) reader this form of aggregation bring together web content from a host of online sources such as blogs, webpages, official websites, video blogs and newspapers to put all news in a central location for easy viewing.

Enterprise also use aggregators at an internal level to help clerical and business users pool information to a single location making it easier to access, and requiring fewer individual programs to be learnt and trained to be used. This single consolidation of business logic and functions can often be found on “portals” a term used to describe applications that use Web pages to display aggregated content from a variety of information sources.

A common tool for portals used within business is the ‘portlet’ which is a pluggable component provided by companies such as SAP, Siebel and PeopleSoft [10]. These portlets act as client rendering and requesting components to a portal server. They request updated data or request to push a change onto one of the portals underlying systems. In a similar way ‘servlets’ are also used to display content for a portal server, only typically these are used to serve complete web pages.

Portlets are not able to generate tags for HTML code, such as; frame, title, html, base or body. Nor do they have their own URL’s which can be accessed like servlets do. Portlets can be duplicated in the same web page and can be manipulated using buttons and controls. Both servlets and portals - which run on Java – are used by enterprise to create interactive web pages where customers and employees can interact with multiple business functionality from a central location [143].

There is, and has for some time, been a wide range of portal server technologies. The word portal is often used to describe gateways, from the Latin ‘porta’ to mean ‘gate’, it also has been used to describe a search engine. This derives from its original usage of sharing information libraries. This might have been institutions aggregating data on a specific topic such as providing citizens of a government relevant information on tax rates, department publications and policy changes.

What began in the middle of the 90’s soon proliferated with the advance of the ‘dot-com era’ (1997-2001) as companies tried to get onto the internet and begin sharing information online, either internally through an intranet, or externally to customers.

IBM WebSphere Portal - By building on top of the IBM WebSphere Application Server (WAS) and merging two of their existing products; Lotus K-station and their existing WebSphere Portal capabilities IBM were able to release the WebSphere Portal in August 2001 in a bid to merge the ability to create the different portal types into a single software.

What IBM were specifically able to bring when they released, was the ability to publish portlets onto the to a web service directory as well as searching the directory in order to add a web service as a portlet onto a webpage. This meant enterprise could quickly create, publish and subscribe to portlets across the business, reusing services and displaying them with greater ease [144], [145].

Oracle (WebCenter) Portal - In 2007, Oracle released WebCenter which included an embedded portal that did not require a separate stand-alone server or runtime instance.

Importantly, it began to see portals as a tool that can be created by business users to suit their specific needs without requiring a dedicated portal developer [146].

This might be where the sign-on service is the same for many applications. Instead of signing into each internal site, it is much easier and more productive for employees to type their password once and also to be able to see all their information in a single location whilst also

improving integration between the two by having remote procedures moving data under the covers.

In a pharmacy, for example, a nurse might receive a series of prescription requests, they can check the internal stock, place an order for new medicine and deal with packaging dosage amounts and storage from a central screen making the whole process easier.

This can also work on a wider level, say, between a car dealership and its partners (suppliers, mechanics). Each partner can allow employee access based on specific needs into the business to business portal. They can then access the processes and business functions permitted to them from an easy to use location. It also allows the dealership to change partners or introduce new partners without needing to spend money creating new integration points.

Advantages: The user interface level is a cheap and easy approach with many products that can be used. The integration can be up and running in a short amount of time and allows less skilled IT workers to use the interface as part of their more experienced job instead of the individual applications.

Disadvantages: The problem with this approach is that the integration is fairly superficial with only a skin-deep level of integration where it can be difficult to keep the systems in synchronisation.

Four Levels of Enterprise

The four levels display the various types of integration that are important to an enterprise, whether that is down at the system of records, communication between two systems, departments sharing tools or a user wanting lots of the applications they use to be easily accessible from a central location. Each level is important and not exclusive.

Enterprise may or may not have high tech, expensive solutions to each of the levels, but the integration must occur nevertheless whether this is a manual implementation, a home-made solution made from bare metal or an out of the box solution.

To give a full enterprise example, it is possible to imagine a newspaper which at the user interface level is using a live feed of aggregated news stories from across the world, at the same time stories being submitted under different categories like health, travel and food are being pushed from the editors computer where it has been approved for publication and onto a collected portal where all approved content is displayed on a customer portal thus displaying application level integration.

At the same time another member of the newspaper is contacting the owners of the images that are being used and issuing payments. This payment method is also used for one-day contractors who come to do maintenance on the building the newspaper owns and for the freelance journalists who have their stories published using method level integration such

as an application server. These payments are stored on a system of record using a message broker to transform the data into the correct format, before the system of record is merged with others into a database warehouse at the newspapers parent company.

The next section describes some key patterns that are, and have been in use by enterprise over the last couple of decades, their principles and their use in every day enterprise life.

Enterprise Application Integration (EAI)

Enterprise Application Integration (EAI) solutions attempt to solve issues in every part of an enterprise; hardware, software, architecture and processes through integration of business processes, applications, data and platforms.

EAI in itself had begun to enter the wider computer industry as a recognised term by 2002, introduced by industry analysts such as Gartner in order to explain the emergence of integration technology [28]. Its original founder, David S. Linthicum the “father of EAI” [33], [34] first wrote a book that was published in November 1999 called “Enterprise Application Integration” which described the concept, the approaches and the pitfalls.

The principles of EAI is to make system integration and processes easier by simplifying information sharing between data applications [29]. This might include the elimination of dependency on specific vendors to allow new alternate vendors to be inserted into the system with little inconvenience to the enterprise.

This allows enterprise to look for competitive software, both in terms of cost and technical ability to reducing ability of a vendor - that has had the same product for the last 10 years with little or no change - from having a stranglehold on a business and their funding, simply because the surrounding technologies are dependent on it.

The integration solutions provided by EAI are collected into dispensable Enterprise Integration Patterns (EIP) with some of the most prolific patterns being created by Gregor Hohpe and Bobby Woolfe, who together designed a pattern language of 65 integration solutions which are still heavily in use today [4].

One example of where enterprise might use an integration pattern is the creation of an API Platform. Traffic will be destined to come into infrastructure via landing zones, load balancers, and a gateway. If the API Platform is serving multiple departments such as human resources, logistics, IT, support and finance. Each new business stream will need to be deploy their own APIs.

Patterns mean that each will need to follow a specific style of API, whether it’s the security of transporting messages, the offloading of APIs to downstream services or even the input and output expected of an API call. Patterns help keep consistency across the enterprise and allows faster and simpler adoption for each new provider or consumer of APIs.

EIPs and frameworks should have the following values at its core in order to stay true to the principles of EAI;
Manipulation/Transformation – providing the ability to transform data or enriching its existing format. This can be achieved through direct conversions from one application to another or by using a common language as an intermediary [30].

Transportation – communication across multiple protocols, product types, software styles and vendors [30].

Mediation - the EIP will act as a broker between the different applications by alerting each component of a state change. It also decouples the applications by creating a separate communication layer [30].

Orchestration – creating clearly defined process flows by organising and arranging the communication of elements within a system in the most effective way; whether automating a process or combining data. Orchestration is best performed by decoupling applications and providing central management [30], [31].

The addition of orchestration in this list may cause some contention, with a discussion between the differences between orchestration, composition and choreography being raised. Whilst similar, they do differ in key aspects. Orchestration describes a central control that describes how systems should pass and use information, in music each system process would be a composition that makes up the business method which is the orchestration.

Choreography has a distributed coordination where each process needs to know how they collaborate with the other processes that make up the flow of data required. Due to no centralised control it becomes difficult to monitor each process and instead each would be required to speak to all the others to acknowledge failure or success.

Where the contention lies is the question of where integration ends and where process management begins. But orchestration, is a central control that integrates processes and services to become a single service and whilst the processes do not make up an integration, the control which is able to integrate the services does allow integration and should be included because of this.

Enterprise Integration Patterns were, and remain to be, a great source for well-defined integration techniques. These patterns continue to be maintained and updated by Gregor Hohpe, incorporating modern integration ideologies into the patterns. These can be found on his blog [32].

Whilst EIPs are EAI patterns, EAI at its core works in one of two ways. Either in a traditional point to point architecture such as Messaging or RPC or as a more advanced middleware solution such as application servers or message brokers.

Hub-and-Spoke

The pattern of hub and spoke uses a central integration platform known as the ‘hub’ which applications use to connect to other applications. These applications make up the ‘spokes’ which connect to the central hub, with all data becoming converted to be compatible with the hub before being processed by it.

‘Hub and Spoke’ was popularised – as a term - after being pioneered by Delta Airlines in 1955 who used it to describe the Atlanta airport where smaller flights would fly to, in order to get connection flights to other locations. The benefits the distribution model brought was a much cheaper method for moving people through the air.

The cost of having direct flights from all locations to all destinations is significantly higher than having a central location for managing, moving and directing passengers to all destinations [149].

Other companies - having seen the model in use - began emulating the structure with the most obvious usage being for the distribution of goods, such as package delivery and shop distribution for global high street brands. Later, this same method proved just as relevant for the transportation of goods purchased on the internet.

In integration, the hub itself consists of multiple models, entailing both the business object models, business logic completion and also data transformation modelling. An application that uses an XML web form that needs to interact with a system of record application that uses SOAP (a type of protocol) will convert first to the hubs desired language, complete all business functions and logic, before being converted into SOAP ready to interact with the system of record.

By centralising the management of business logic and ensuring each spoke input is first converted into the format of the hub, and then is able to convert back into its required output format, means that each spoke need only know a limited number of conversion formats, making each application decoupled and only having to communicate with the hub.

Whilst the hub requires a lot of development, it is not necessary for each enterprise hub to be made in-house and can instead employ many of the technologies discussed in the enterprise service bus section, much in the same way to perform transformation, orchestration etc.

Hubs also benefit from reuse of existing connections. If the hub is already able to convert from one format to another, the same transformation can be used for a new entity wanting the same transformation.

The obvious disadvantage to a hub-and-spoke architecture lies in the limitations of physical hardware capacity, which has a finite number of possible messages that can be placed through it before requiring an increase in size in order to scale. From a technical point of

view, in the land of infinite time and money, you would build a highly available hub and spoke system that can handle the enterprises message load in its entirety as this makes provides a central location to perform management, monitoring, maintenance and governance.

However, scaling on the hardware level is an expensive method, as message throughput changes over time, i.e. an umbrella company seeing more sales during April Showers than a hot August summer day. The time taken to spin up new hardware can be months or even years in some organisations and just as long to tear them down or more inconveniently - sat idle.

To avoid this situation hub-and-spoke architecture patterns can be federated - a series of hub-and-spoke architectures that intercommunicate either through a central hub, with other hub-and-spoke's being the central hubs spokes, or through a series of hub-and-spoke architectures that intertwine in a disorganised manner. Each of these hub-and-spoke architectures would have isolated services, rules and business logic.

Whilst this makes logical sense it leads to the issue whereby business functions (logic, rules, services) need to be replicated across each 'Spoke' where they are needed. In turn the problem arises in that business functions may be on different versions, each architecture may be reliant on external factors like virtual machine, network and underlying software.

One method to avoid the replication of business functions is to use a lookup within the architectures. By first checking internally - within a spoke's own hub. A request can be made to find the required service amongst the federated hubs. The hub of the requesting spoke can query its parent hub - of which it is a spoke - which will then check its children hubs. Once the required service has been found, the requesting hub can communicate directly with the service owning hub.

The hub-and-spoke architecture leans heavily on modelling data and data object to more easily move data between software and machines.

Data and Object Models

A business object is a set of interface definitions to give a specification on data structure, data format, message type, elements, semantics and syntax. Additionally, interface characteristics such as; size, transactionality, availability, reliability and so on are dependent on the APIs and Services themselves.

There are four main types of business objects in the Hub-and-spoke architecture, they are; GBO, ASBO, SEBO and APIBO. It is worth noting that the usage of the terms varies between enterprise and that a GBO used by one business may be referred to as an ASBO in another and the context of the application of the object models defines how each data set fits.

1. *Canonical Data Model / Generic Business Objects*

Canonical Data Models - which are sometimes known as Generic Business Objects (GBO) - reside on the Hub to determine a set of data specifications that will be used when messages travel through the hub.

Each spoke has its data points mapped to that of the GBO which allows enterprise to work with the data formats that fits best for them, whether because of knowledge base and skills within the enterprise itself or to conform to other data types used in other applications and processes.

A benefit of the GBO is that changes to data models only need to be made at the GBO level and to the maps created to connect to other applications and there needs to be no communication with external sources.

There are some notable concerns with the Canonical Data Model usage in that to apply these standards across an entire enterprise, and force a single business object on all users, teams, processes and services disregards the need to take each as a separate entity with its own requirements.

Instead each new process should be able to describe new business objects that work easily for *them*, whilst business objects should be adopted and not enforced.

2. *Application Specific Business Objects*

Application Specific Business Objects (ASBO) resides on business applications to define the structure of the data used. Processes, services and business flows will use adapters to map the data provided by the ASBO to the GBO.

ASBOs are defined by the owners of the applications that the process is connecting to, but it is the process owner who maps the two data points together. If the GBO was to change, the mapping needs to be altered, not the applications themselves.

3. *Service Exposure Business Objects*

Service Exposure Business Objects (SEBO) describe a collective data format that a community of service members agree to use in order to make integration easier and more quickly replicable.

It requires each part of the data format and interface characteristics to be agreed upon - often contractually - with any changes needing agreement from all parties. This means that SEBO implementations need proper version control, documentation and release timelines.

4. *API Business Objects*

API Business Objects (APIBO) are stateless, RESTful API data format definitions that are community driven to provide access to enterprise data and services to make useful

applications. APIBO's are well documented, easy to use and have a low barrier to entry.

These business objects form a very important part of the Hub-and-Spoke design as enterprise try - and succeed - to make central management of integration easier to maintain and expand. In systems where applications dictate the data types used by the GBO due to product limitations or even skill limitations within a team, the result can be a complex system which is constantly changing due to high importance consumers.

Additionally, complexity can be found when the GBO changes. This might be due to regulatory needs i.e. the introduction of a new currency or elimination of a currency denomination, with consideration being required for how the migration of data changes should be applied. For instance, should the GBO set a defined date for migration, or should It facilitate both old and new data inputs? Should altered data types be set by well documented default based on consumer analysis, or should a requirement be set for the applications to set the data type with call failure if not inserted?

Decisions, when made, need to be described to all the connected and affected Applications who may need to update supported software or alter custom connectors in order to conform to the new GBO needs. Connections to the hub itself may number in the thousands, and when larger federated architectures are considered this may even be hundreds of thousands of applications.

Everyday Hub-and-spoke

To give an example of a full hub-and-spoke setup, an online game company can be considered, which provides web and mobile applications. It connects to a payment application for in-game purchases and a cloud-based game play data storage service.

The application server contains a process for accepting payments, which causes an increase in usable items in the game play data storage. The process itself uses a GBO that is a simple data format of JSON with easy to read and understand names such as; Customer ID, Customer Name, Game Statistics, Purchase Statistics. The GBO caters to the in-house development teams Markup preference and allows better supportability of the process.

A customer makes a purchase within the online game, using a world-wide payment broker which returns a lengthy XML response to the process. The ASBO uses abbreviated titles with underscores e.g. <CUS_NAME/><CUS_NO/><ADDR/>. Additionally, the cloud-based data storage which is used to track game play and paid for content uses an SQL framework needing a connector to communicate with SQL fields e.g. CUSTHDR.CUSTNAM, CUSTHDR.CUSTNUM. Both of these ASBO's are mapped to the process GBO.

The mapping is the responsibility of the online game company which the application server maps all application objects to, which means the ASBO providers do not need to make any changes to their objects. The onus of the mapping is entirely with the online gaming

company meaning the data is moved internally in a format the company is comfortable using.

After a short time, the online gaming company, the cloud-based data storage provider and the payment service determine to use a common format. They discuss and debate which data format, delivery method and naming conventions the common format will have which they finalise, version, publish and implement throughout. Each new process that connects the online gaming company and the applications will use this newly defined SEBO.

When the SEBO is introduced, the company maps it to an employee bonus process for in-game purchases which has a separate GBO as it is ran by another department.

The game later gains a lot of success, with third party companies beginning to use publicly accessible data to provide users of the game with game management services such as timed in-game interactions and automated in-app purchases. They request a more formal arrangement of access to the game data. This leads to a community driven, community agreed set of definitions for APIs, security etc. (APIBO) that the gaming company agrees to introduce by a certain date and free of charge.

After an initial six months the company begins to monetise the APIs, introducing rate limits to charge for heavy API usage but with business logic to charge relative to the size of the company. These internal APIs might be APIBO's or ASBO's within the growing hub and spoke architecture.

Collectively this gives an enterprise example of business objects, the architectural view of a hub, its spokes and a spoke which is itself a hub.

Enterprise Service Bus

The Enterprise Service Bus (ESB) is described by David Chappell in his book "The enterprise service bus" in 2004 as "...a standards-based integration platform…". The platform will combine a multitude of concepts to allow diverse applications across an enterprise and its partners to reliably connect. This coordination is possible by using the concepts described previously, such as; messaging, data transformation and routing whilst also providing transactional integrity. Additionally, Chappell included web-services in his description which will be discussed in more detail in later sections [58].

The ESB is known as a bus as it connects all the applications from a central hub. It receives data from applications, performs some functionality on the data such as transformation and then delivers to another connected application. The bus analogy comes into play when the architecture is drawn out to show data passing along a "route" with connecting applications providing the bus "stops".

From a historical point of view, the originations of the term ESB is accredited to W. Roy Schulte from a 2002 Gartner report called "Predicts 2003: SOA is Changing Software" [59].

However, there has been much debate around who first coined the term ESB and its concepts. Vivek Ranadivé the founder and once CEO of TIBCO asserts that TIBCO “wrote the book” of ESB’s with TIBCO’s own name (The Information Bus Company) as proof that they had used the term first, though Schulte says the first time he had seen the term was at the release of Sonic XQ by Sonic Software in 2002.

When considering the term bus, earlier core computing concepts had utilised the term to describe how information is passed in a system. One example might be the processor needing to talk to the memory. In order for the memory to be read at position 1234, the processor needs to input the value it wishes to lookup onto the address bus, which uses a set of physical wires (32 bit) before sending the information back via the data bus as 32 Bits.

The term bus was later applied more widely in computing - including in integration - to describe data connection between two or more entities in a computing platform. In some examples the ‘bus’ is found within a computer, in others the ‘bus’ is more broadly across multiple connected devices.

This use of the phrase bus was used in messaging applications as a ‘message bus’ before a wholly new concept - Enterprise Service Bus - tried to describe the exposure of synchronous web services. Whilst this may have been its initial purpose it has since been distorted by varying implementations and organisation level branding and terminology translation to mean a much wider periphery of connections encapsulating all forms of integrations.

In reality there were many companies that had true ESB-like software throughout the 90s, with evolving topologies naturally gravitating to the benefits that a centralised integration hub brings. The cost of maintaining point to point integration was becoming unbearable to business and as such integration architecture was evolving in that way.

TIBCO could boast business process management (BPM), transformation and adapters from their own product and TIBCO had actually espoused in their product what most early ESBs did not by including these features. However, by definition an ESB should support web services which TIBCO did not [60].

During the same period, the company Candle had released a product called Roma, which along with CrossWorlds Business Process management was instrumental to some of the technological ideas for the ESB. In doing so they became a prime target for larger integration companies and in 2002 and 2004 respectively, CrossWorlds and Candle were acquired by IBM [185], [61].

Whilst IBM had other integration products that delivered key ESB functionality, none could be seen as a complete ESB solution as they each did not include at least one of the key components of an ESB.

Despite this both TIBCO and IBM had initially argued that their existing products were true ESBs, though industry adoption of service-orientated architecture (SOA) soon showed this

not to be true. TIBCO released its first true ESB in late 2006 in the form of BusinessWorks 5.3 [62].

An ESB consists of mediation and service composition and can be broken down into these components.

Mediation:
The mediator is responsible for message transformation and processing, protocol, security and routing.

In one example, two customers purchase one item each, one from the US and the other from the EU where regulations state all purchase data for this kind of transaction must be secure. The mediator will route traffic based on where the request originated and will ensure the purchase is completed over a secure protocol with the EU data encrypted.

Service Composition:
This component of service management defines services at a global level by establishing interaction rules and message exchange agreements between two or more endpoints.

Historically, the term ESB has been used to encapsulate a number of different technologies, solutions and architecture with many companies taking existing stand-alone integration products, merging, slightly improving and then releasing them as ESB solutions. Whilst this helped get lots of more advanced integration packages to enterprise, many of them missed one or more of the key components that make up a true ESB such as web services or service composition.

The fallout of calling all products an ESB whether they met the requirements or not, was a confusion of what an ESB truly was and still is. In some existing environments the ESB will be loosely aligned to Integration Buses; both integration bus solutions that will conform to a true ESB and those that do not. In other areas the ESB is seen as an integration box whereby all message and applications flow through.

Whether a system is a ‘true’ ESB or not, the primary benefit of the ESB approach - even partial - is that it makes changing major integration applications both simpler and easier because it reduces the number of point-to-point connections the system needs to allow communication across applications.

ESB’s give a standards-based approached with many pre-defined services which detracts from a code heavy implementation towards configuring software solutions for specific applications. This might be an enterprise having an ESB that can map all XML formats to a corresponding SOAP structure, each connecting application will map their XML format in the bus using some form of XML to SOAP translator before sending the message via the chosen route.

To map each applications XML directly to each consuming entity directly would create exponentially larger webs or meshes of integration points. Which is harder to maintain, requires more customised code and makes assets less reusable and thus each new integration is more expensive.

But like all architectures and patterns, there are some notable concerns. For example, for an ESB to be affective it needs to be able to support a wide range of application connections which may have their own constraints on data formats such that it means a large quantity of data forms or connectors are needed to allow integration through the bus all of which need maintaining.

Another consideration is that the ESB is upgraded as a monolith in that everything must be upgraded at a time. Whilst individual integration teams and connections may work separately, they are tied together to the infrastructure in a way that means they must all upgrade or none can. This comes to ahead if new function is necessary for one team, but a period of stability is necessary for another.

In the same vein, ESB platforms would be managed by a centralised team who control the governance for the product across the wider teams. In this world, control isn't federated which makes it difficult for teams to be independent and thus are tied intrinsically with core timelines and dependencies.

As with any platform, if not correctly implemented issues can be caused with scalability where some of the typical products that fit into the ESB platform simply do not scale well. This is primarily because the ESB becomes a bottle neck that additionally cannot be easily increased due to the dependence by other teams.

Service Integration

What is a Service?

In integration, as in the real world, a service is "the action of helping or doing work for someone" [63] and in computing these refer to pieces of integration that allow disparate systems to connect through small pieces of callable work.

Services are Application Programming Interfaces (APIs) that allow users to piggyback on other application's exposed functionality. These APIs previously would have been understood as the entry points between systems using such functionality, as used in RPC or those used by java and other software languages.

The first web services were launched by a company called SalesForce at an IDG Demo 2000 conference before being followed shortly after by eBay in November 2000. In 2002, Amazon

launched the Amazon Web Service (AWS). However, it wasn't until the social era and the launch of flickr's first RESTful API that web services began to explode into the market.

These APIs allowed third parties to quickly create applications that enhance either the users experience of the API Providers services or to capitalise personally from new services provided by the third parties [65], [66].

Web services - as previously described - are a form of API which stemmed from RPC models in Distributed Computing Environments (DCE) which was, and remains, a 'C' leaning language with a platform neutral client/server architecture. The benefit gained from DCE for web services is the Interface Definition Language (IDL) document - the service contract and input used as an input for utilities generating artefacts in support of the DCE/RPC calls [70].

By 2004, W3C had defined a web service as a software system with the following characteristics [75]:

- Interoperable interactions over a network for machine-to-machine communication
- Machine-processable interface such as WSDL that define external communication
- Typically use SOAP messages
- Typically conveyed using HTTP
- Uses Web standards
- Makes use of XML serialization

Service Maturity

The way in which enterprise integration service maturity is determined is through neutral SOA levels defined by the Open Group Service Integration Maturity Matrix (OSIMM) [101]. The seven levels work to show where an enterprise is in its maturity without prejudice i.e. seven enterprises each on different levels might be at the level that makes the most sense for the enterprise at the given time.

Level 1: Silo

Separate parts of the organisation are building software independently – or as Islands – There is no integration of data, processes, governance, standards or technologies. Integration requires manual intervention to proceed.

Level 2: Integrated

Light levels of point to point integration connects disparate systems allowing the integration of data. There is still no standardization of data and connectivity depends on purpose written code, adapters and protocols making it difficult to develop and automate business process.

Level 3: Componentised

At this level there is some standardization coming into place through Enterprise Application teams. There has been a breakdown of current applications into components with a framework for new integration systems. Components are reusable but are often replicated and can quickly become redundant. They are not loosely coupled which can cause issues with interoperability.

Level 4: Services

Companies have begun defining SOA paths taking into consideration the needs of new projects before they are implemented. There is an IT infrastructure that is supported by a framework for data transformation, protocols, governance and security. Services may or may not be interoperable across all parts of the enterprise. Services are defined through by a specification language such as WSDL though this will still rely on purpose-built code for composing applications.

Level 5: Composite Services

Organisations at this level have begun defining business processes using existing services using a mixture of bespoke code and composite or business process modelling of information. This level uses a static, process and activity-based services. Services are now “agile” and performed by business analysts and developers.

Level 6: Virtualised Services

Services are called virtually without direct invocation which allows a loose-coupling integration of services as the infrastructure works to turning virtual service calls into physical calls where address, network and protocol are adapted by the infrastructure.

Level 7: Dynamically Re-Configurable Services

This level of service integration sees runtime services configured to the needs of a business process. Either through pre-configured services stored on a repository which can be accessed and queried by the required services [92], [102].

SOAP

By the late 1990’s XML had gained notoriety and widespread use mainly due to it being readable by both machine and human. This meant that different machines could talk to other machines through human applications in a format easily recognised by all.

Hence in 1998 Dave Winer of the UserLand Software company created the XML-RPC protocol alongside Microsoft to complement their applications, with Microsoft opening up their operating system using the technology [71].

This would later become Simple Object Access Protocol (SOAP) an XML-RPC protocol across the HTTP architecture, with the help of DevelopMentor. The three companies had previously worked on separate interfaces but began to standardize towards the initial SOAP protocol. The protocol, however, wasn't shipped until late 1999 despite the initial protocol work, apparently, prevented by internal Microsoft politics [72], [73].

By 2000, Microsoft and competitor IBM had developed two separate web service definition languages. Network Accessible Service Specification Language (NASSL) for IBM and Soap Contract Language (SCL). Whilst the two worked almost identically on a functional level, the two languages had different vocabularies for XML information representation.

Thus, the two joined forces to bring the two language types together, forming a unified description language called Web Service Description Language (WSDL) in September of the same year. Over the next few months the two continued to refine the language, publishing the WSDL 1.1 specification to W3C in March 2001 having amassed the then largest submission team in W3C history [74].

REST

Representational State Transfer (REST) a dissertation by Roy Thomas Fielding from the University of California in 2000 which revolutionised the sending of data between two applications [64].

REST was designed to rely not on XML but instead focuses on using a Unique Resource Locator (URL) alongside the four main HTTP 1.1 Verbs (GET, PUT, POST, PATCH, DELETE) and a defined media type in order to perform tasks.

Unlike SOAP, REST is able to output in a variety of structured language forms such as described in Integration Considerations sections. They each have their own advantages; REST requires a HTTP point-to-point which makes it easier to use as it doesn't need expensive tools for interaction. SOAP on the other hand has built in error handling; language, transport and platform independent and works well in distributed systems [81].

Mr. Fielding described RESTful systems as systems that send messages which are either at REST or transitioning from one RESTful state to another. Each component is bound to obey the simple set of rules prescribed by REST [80].

Within REST there are guiding constraints - that when followed allows a service to become RESTful and so should in theory meet the non-functional system requirements of any system that uses them; such as scalability, reliability, portability and potential for modification.

Uniform interface principle denotes separation and independence of the resource accessing data for the REST call. This is provided by allowing the sending of data from a resource system in multiple common formats that are not necessarily the same format used by the resource itself.

Representation through a uniform interface should allow the consumer to perform manipulation of the resource (update/delete), the consumption of the resource by interpreting descriptive messages, and possible additional actions/resources should be discoverable via Hypermedia as the Engine of Application State (HATEOS) i.e. a hyperlink.

This independence is best shown through the client-server architecture whereby the client interface is separated from the server so that the two are able to function without interdependencies. The client sends and requests information to the server in a set format but the two can be scaled, upgraded and worked individually.

As part of the interaction between the client and the server all information must be described within a single request and cannot rely upon context it might happen to know is stored on the server. This stateless constraint requires all session state to be stored client side which might cause some performance trade off due to the amount of data relayed between the two systems.

The overhead that is brought about storing context solely on the client through increased interactions can be reduced by stating whether response data is cacheable or not. By using cached data, the server is able to explicitly say what can be reused by the client again although stale data can quickly become different from the expected data provided by the server in a fresh transaction.

A further constraint implements a ‘layered system’ whereby each component can only see its own layer in a hierarchical architecture. This means it makes calls above and below it. This allows for further decoupling of the system by embedding a system of each service not concerning itself with the full stack flow making it easier to evolve and reuse each service in the architecture.

Whilst a purist would determine each constraint must be met to conform the RESTful practices, many systems do not. This can be for a whole host of reasons ranging from product constraints, trade off decisions e.g. not wanting the overhead of not always using a cache, and in-house skills. For this reason, REST was largely ignored in the years following its publishing and was only implemented with a passion as enterprise moved towards the API economy.

Service-Orientated Architecture (SOA)

The term SOA gained widespread popularity and adoption after it was described by the Gartner Group in 1996 in a report entitled “Service Oriented” Architectures, Part 1 by

Schulte and Natis [87]. They further discussed the design and implications of these architectures in “Service Oriented” Architectures, Part 2. Before going on to write an introduction to the practice in 2003 [88], [89].

Prior to that Schulte himself has accredited Alexander Pashik for the terms first usage during a class he was teaching middleware to in 1994. Pashik’s motivation stemmed from the loss of the classical ‘client/server’ meaning which was widely being used to describe distributed computing with both the “client” and “server” hosted on computers. To avoid confusion “server orientation” was stressed as he described SOA business applications [90], [91].

But the definition of SOA became a stickier point with many variations. David Linthicum in 2008 described SOA as a strategic framework for exposure of services that can be abstracted for orchestration [92]. Whereas the Microsoft website – at the time of writing – describes it simply as a loosely-coupled architecture to meet business needs [93]. The Oracle website – at the time of writing – ties the two definitions together to an extent, describing an architectural style for building applications using loose coupling of services that are available on the network specifically for enterprise reuse [94].

There are a whole host of definitions in the book “SOA in Practice: The Art of Distributed System Design” By Nicolai M. Josuttis, which serves to prove how varied and dissimilar the industry sees and uses SOA in enterprise.

The main point behind SOA systems was that it was the first time that the method for calling web services had been standardised at an industry level. Standardisation itself was not a new concept. The industry had previously known component object concepts and interfaces through both Common Object Response Broker Architecture (CORBA) by the Object Management Group and through Microsoft’s Distributed Component Object Model (DCOM) [95].

DCOM was created in 1996 and is windows compatible, stemming from Microsoft’s previous framework of Component Object Model (COM) which was released in 1993 and was preceded by Object Linking and Embedding (OLE).

The initial benefit of OLE was the ability to create and maintain compound documents using embedded or linked objects which could then be implemented very easily and quickly. It wasn’t until the release of OLE 2.0 in 1993 that the technology began to open to a much wider selection of component software [96].

CORBA began its life in October 1991 when the Object Management Group (OMG) – a not-for-profit technology consortium – released the first version of the object modelling technology. Whilst initially not interoperable and providing only a C mapping it gained numerous features by its 1997 CORBA 2.0 version which allowed developers to build heterogeneous distributed applications [97], [98].

SOA became prominent as web services grew by being the simplest approach for implementing loosely coupled architectures for the use of integration. Though this did not mean less usage of CORBA, DCOM, and Electronic Data Interchange (EDI) – a system which had been used for standardising documents shared between enterprises – which have grown and adapted alongside web services though are now being over-shadowed by the microservices approach.

As a misunderstood philosophy, SOA is neither a technology or a methodology and cannot be built. It is not new, does not require Web Services to be implemented and should be built on top of existing investments [93].

One of the greatest benefits of SOA is that it encourages the reuse of everyday software assets to reduce repeating business application functionality and instead uses "loose coupling" to allow the same services to be reused for many tasks that would usually take a long time to annotate for each business need [99].

An example would be a service for checking the balance of an account. This may be used in multiple environments and across different processes. The balance checking service would be reused for multiple larger services such as: Account Enquiry, Loan Validation, Bank Transfer, during a purchase etc. SOA looks to reuse the service instead of writing it fresh for each business process.

Service Orchestration is a vital part of SOA where components can be strung together or connected in order to allow a business process to be completed. This is made possible by making services dependable and to a well-defined level [99].

Two components that are used within SOA are the service registry and rule engine which are used to in a wider SOA context to gather services together or determine how to route requests as they enter the system.

Service Registry:
This component deals with service orchestration – determining the direction of a web service business process through a central controller which may or may not be a web service in itself.

For example, employees might land on a corporate webpage to log their hours of work for the week, however each user is assigned to a different organisation within the corporation and so the service registry directs organisation specific traffic to their relevant services.

Rule Engine:
The rule engine is also responsible for message routing but can also be used to determine how and when message transactions should take place as well as providing message enhancements where applicable.

One such process that might require a rule engine, is a process that directs overtime

payments to employees with the payment amount determined by the date and time of the day overtime was worked and the type of employee working the overtime.

In order to best expose these services, there is a need to standardise protocols, data forms, component visibility for purpose of monitoring, traffic management (for down time, prioritisation and accepted response times), virtualisation in terms of routing and data translation and a coordinated approach to service security [100].

Whilst these can be agreed principles of SOA there is variations of implementation due to the initial immaturity of the concept making it a different thing to the different enterprises that adopted it, who have themselves began to develop SOA to their specific needs. This is well described by David Linthicum who stated SOA adoptions being "...like snowflakes - no two are alike.". No two SOA implementations are the same and with a batch message processing system being called SOA equally to an employee benefits portal [92].

In truth each organisation adapts SOA to their specific needs, they make different levels of integration in regard to their size, IT needs and maturity of the enterprise.

Whilst the topic of SOA is contentious and broad in its scope, it is a vital way in which business and enterprise plan and invoke their integration needs and will continue to do so alongside and in harmony with microservices.

APIs

Initially, there were little to no APIs to allow access to the data, forcing 'bedroom developers' and new start-ups to use data scraping to gather publicly accessible data, creating "Frankenstein" APIs before companies themselves released versions in 2006 [65], [66], [67].

Further evolution was to follow, with developers beginning to create web applications on top of existing code to allow for public consumption. This type of exposure was expanded from a single public API in 2005 to exceed 15,000 at the start of 2016 [68].

A key driver for the exponential growth of APIs over the same period stemmed from the ability for small developers and small companies to emerge quickly, in collaboration, and gain a joint profit with larger enterprise to open new markets and 'disrupt' existing markets.

Package holidays, hotel and flight comparison sites began using APIs to give customers wider choice in prices, the ability to filter and sort offers provided based on customer needs. The comparison sites gained commission, the industry stayed competitive, customers got the best deals available and all of this was available from the customers' homes and phones, with new deals provided in moments.

Large amounts of the growing demand for APIs is made up of web services which must be available over the internet, must be independent to any specific operating system or programming language and use a standardised messaging system alongside XML [69].

Between 2003 and 2007, companies such as; Myspace, Facebook, Tumblr, YouTube and Twitter had emerged to allow users to interact through the sharing of their photos, thoughts, videos, likes, dislikes and feelings. With this vast increase of content being uploaded onto the internet, the sites soon held vast amounts of data on users, that would turn out to be key to the way enterprise across all industries operated and became influential in enterprise decision making and as such, their associated applications.

Agile Integration Architecture

An encapsulation of cloud applications using microservices, 12 factor application principles and the “cattle” approach is Agile Integration Architecture (AIA) which looks at both where an enterprise is in their ‘journey’ in terms of integration architecture maturity and also the steps required to get from an existing stage in the process to the most advanced.

Whilst originally called ‘lightweight integration’, Kim Clark - an IBM integration specialist - expanded and adapted the concept of simply containerising existing systems in a ‘lift and shift’ approach to rethinking entire integration architectures for the purpose of providing agile, scalable and resilient integration systems.

In AIA there are three core aspects that lead to a final Agile Integration Architecture from a traditional centralised ESB or even an isolated island of integration with individual connections to other systems.

Three core aspects are used to describe how AIA can be achieved which relate to a move to containerisation, application autonomy and polygot runtimes.

Aspect A: Fine-grained integration deployment

By breaking out individual integrations from a monolithic application into containers of integration applications there is an ability to independently scale and manage connectivity between different applications and services. An example might be a software that is used to connect a system of record with payroll and an overtime compensation system.

If the monolith is scaled because it’s the end of the quarter and employees are pushing for additional sales by working extra hours, there is an increased need for additional capacity to accommodate the overtime management. When scaled in a monolithic system this causes both the payroll and overtime integration services to be scaled together rather than for the need of the individual service.

This reduces cost because the containers are sized according to the services requiring the increased use, whilst also giving the benefit of an overload of the overtime containers having an adverse effect on the payroll system.

When containerisation has been achieved the system is now described as a ‘fine-grained integration deployment’.

Aspect B: Decentralized integration ownership

Having now containerised, the ownership of each of the runtimes can be passed to individual teams who are best placed to understand the need and cost of their services. This ownership extends to the management of maintenance, operation and creation of runtimes.

There are additional benefits to be gained from decentralising in that not only is there an increased efficiency of runtime management because teams are handling their own needs but there is a reduced need for the wider organisation to understand each runtime which reduces the cost of handover and knowledge transfer.

Aspect C: Cloud Native Integration Infrastructure

The final step towards AIA is to take integration code that runs within applications and services and deploy various pieces of independent code into containers to be used in combination to complete more complex integration runtimes and structures.

This further separation, isolation and decoupling provides an extra level of scalability and reliability to the architecture as whole.

Considerations

As with all containerisation and microservice practices the same considerations continue to exist around state, granularity of isolation and decoupling of runtimes across vast enterprise systems [158].

Simply putting existing applications into containers doesn’t mean that all the problems of a monolith will be resolved. Primarily, a simple ‘lift and shift’ of applications from bare metal to containers might not be best adapted to get the benefits of containers, might have poorer performance than a ‘bare metal’ deployment and may struggle to communicate with other systems.

There is also the concern of multiple containers sharing the same system and therefore the same kernel with each typically having root access. This poses a significant security risk in comparison to systems with greater isolation such as independent virtual machine deployments.

A less discussed consideration is the governance of each of the containers, runtimes, integration points, management, deployment and alterations. In the past where system owners would have full control over the users, groups and access control of software, servers and applications a transition has occurred whereby DevOps owners have greater levels of controls over live applications and services.

Whilst decentralized governance is important, there are some occasions where a centralized system is beneficial. One example might be a shared service repository which is better funded by a wider enterprise. This requires some level of centralized management to ensure the platform serves all its users using the platform.

At a higher level the system owners themselves have a much-reduced role with access given to either DevOps teams or to container management owners such as helm and Kubernetes administrators. Whilst not a concern in that governance still exists, it does require thought into the control of deployment of all elements of the new integration architecture.

Further thought must also be given to the monitoring of integration systems in order to ensure the correct number of instances are running and are active as well as a greater reliability in persistent storage for containers and the container management systems themselves.

Implementation Considerations

Hosting

Mainframes

Originally used to describe the physical aspects of a large machine we might instead call a Server or Computer today; the mainframes demise has been predicted since before even the early 1990's when IBM saw quarterly losses of $8 billion. With the rise of the personal computer, many saw the mainframe as obsolete and investment in mainframes over PC's fell, with the price of mainframes dropping by up to 90% [110].

But mainframes are making somewhat of a comeback. These 'big iron' computers actually work well with modern day application building practices; they also continue to provide amazingly quick processing speed whilst being prolifically secure and reliable.

Companies typically use mainframes when dealing with large-scale transaction management, the hardware also utilises virtualisation which allows a server to fork its current operating system allowing deployment of compartmentalised sections of an operating system for a specific purpose.

Mainframes work by segmenting a mainframe into virtual systems that act as containers for applications and in this way, it can provide fast and simple connectivity and integration for the applications and services it is supporting. Within the mainframe itself it contains all the key components that allow it to share data, either by partitioning shared memory or by transferring data using messaging and file transfer. This means files can be quickly transferred between one application and another in a setup that already has all the key components it needs.

Whilst these large machines have declined in fashion, they have continued to evolve and stay relevant, whilst remaining backwards compatible. A perfect anecdote for this is best seen in the IBM *Customer Information Control System (CICS)*.

In 1968, in a California laboratory, IBM created a temporary product to help it deal with competitors who were building products that managed mainframe application and server connectivity and transaction. It was intended to last only a few years whilst a new product was designed and built, but the product continued to add value across enterprise and continues to run today, 50+ years on.

To do this, it consistently added new features for the end user, for example; it became a command level system in 1976 running on Virtual Storage (VS). In 1987, CICS was released on Multiple Virtual Storage (MVS) an early precursor for *Z Operating Systems* (Z/OS) which was announced in 2000. More recently it has added Java support and integration with

automated delivery pipeline software such as Jenkins and Urban Code Deploy. The most recent updates being provided in 2017 with IBM MQ integration, asynchronous CIC API and flow control at start up and shut down [12].

By staying relevant, mainframes continue to dominate in enterprise where it still used by 92 of the 100 banks, handles 87% of all credit card transactions, along with 29 billion ATM transactions and 4 billion passenger flight bookings annually [111].

Containers and Cloud

Mainframes excel in the transaction space and can be utilised for applications and services. But a lack of technicians to use these systems, and a strong desire for enterprise to be able to create enterprise solutions in a box which are quickly scalable; up, down, left and right has led to an increase in demand for both containers and the cloud used in parallel.

In 1979, UNIX v7 saw the development of the chroot system call that allowed filesystem isolation by Bill Joy, at Computer System Research Group at UC Berkeley. It wasn't until 1998 when chroot was extended as part of a funded Berkeley Software Distribution (freeBKD) project that looked to solve the problem of having multiple systems running multiple versions of operating systems, software, databases etc. on different machines.

The main issue with running these systems on multiple machines meant that each lay idle for most of the time, since, they were only used when a specific project was being worked on. During the course of a series of emails between a Mr Derrick. T. Woolworth and Poul-Henning Kamp, Kamp agreed to work on the project and give exclusive rights for one year to Woolworths company.

Development worked on five key areas to limit process visibility, to isolate the resources from each other without the ability of breaking the 'jail', assigning IPs to each 'jail' and deciding what the super-user ("root") for each jail could do [112], [113].

Linux vServer had begun Operating System virtualisation by 2001 but Googles 2006 research and development of process containers (control groups) by Rohit Seth and Paul Menage led to greater development of these containers. What they realised was that containers had a need to limit the amount of resource available to each group, that users needed to be able to monitor container usage and prioritise groups CPU usage and I/O throughput [114].

Increasingly, the developments in the container space have been ever more about the management and control of containers to perform testing and scalability. In 2011, Cloud Foundry Warden developed a method to manage containers across multiple hosts as well as cgroups, namespaces and process life cycles. In 2013, Google released their 'Let me contain that for you' (lmctfy) which allowed application to become 'container aware' and thus be able to create and manage their own subcontainers [115].

Just a year later, in June 2014 the friendly blue whale that stands as an icon for *Docker* was released as an open source container project. Developers could quickly create applications by using the Docker Hub to source workflows and content on the cloud and its standardised Docker Engine. These containers were ready for enterprise use, widely compatible, easily extended and could be deployed and run on servers, laptops and the cloud [116].

What makes containers so compatible and beneficial on the Cloud is that they are reusable and portable. They can easily be deployed in the Cloud with large jobs run in parallel across a Cloud cluster, but also easily reused for new processes when there is no traffic to the applications. Containers on the Cloud allows for maximisation of resources and utilises outsourcing of hardware and networking to companies offering Cloud subscription for a fee.

This ease has worked well over the last five years with enterprise and individual developers quickly making the most of the technology. Google, for example, runs 'EVERYTHING' on the Cloud [117] reportedly deploying billions of containers every week. The BBC as another example, were able to deploy test environments to Cloud Docker containers in less than a day. Previously, developers had been waiting up to and over an hour to queue a single job because jobs were deployed sequentially and eventually testing was done more locally with mismatched versioning. Containers meant every developer could test in a replicated and standardised environment at ten minutes each.

Container Management

With containers created and including a full application and the elements (libraries, binaries, etc.) it needs to run, enterprise is able to leverage the technology to redeploy the same applications without manual build and harmonisation with the existing environment.

Enterprise, for example, that wish to upgrade the version of an underlying application version to include some new feature released by a software laboratory, can deploy the container with the same deployment configuration but with the software's new version container image, this can then be tested before slowly introduction production level traffic to the new container to further test the new version.

When satisfied, the old container can be suspended and removed from the environment. In this way - upgrades can occur with little disruption to the system whilst the original container is ready to be reused should anything go wrong, until the enterprise is satisfied with the new container.

Whilst this decreases the risk of human error the system still requires management and a team or teams to do so. With containerised systems, there is a need to manage the base image operating system, the applications which makes up the image, the customisations to the image for the environment it is being used in, the required libraries and binaries, deployed artefacts to the container, the container itself and the container management system to include; load balancing, delivery pipelines, network configuration, high availability and disaster recovery.

The resourcing issues, the CPU usage, the need for specialists for each of the deployment stages and an understanding of how systems deployed to multiple clouds work and continue to provide enterprise production level of service have not been eliminated with the increased usage of containers and are still a key consideration for all integration projects.

For developers, consultants, business management and business leaders this leads to a need for a wide range of skills in every individual as well as the technical competency to understand concepts outside a classical product specialisation. This can lead - if not managed properly - to a workforce who does not fully comprehend the services and software being used and therefore cannot fully take advantage of any of the products in the stack.

To help with some of the more intricate and manual deployment requirements such as high availability, load balancing, traffic management and networking. Many container management systems exist, with the most famous being Kubernetes (K8s).

Kubernetes was created by Google in 2003 and 2004 as a small-scale project to manage a multitude of deployed jobs across many applications and clusters to allow large-scale cluster management. Initially called Borg, the system was followed by Omega before Kubernetes was finally released as an open-source release in 2014. By mid-to-late 2014 other companies have joined the community including IBM, RedHat, Docker and Microsoft [159].

Together the industry expanded the core functionality in order to fit customer and industry requirements for faster, consistent and reliable cluster management. This led to a scramble to deploy existing software solutions into container architectures using Kubernetes.

To enable users to quickly deploy Kubernetes clusters, charts can be deployed. For K8s, Helm charts describe the environment, the container cluster should maintain, how many replicas, the load balancer and network configuration and the public or private repositories for these images.

Helm was developed to allow repeatable deployment of Kubernetes architectures that can be searched, found, installed and managed easily. Helm Charts can be created and installed from container image repositories to ensure the number of container replicas and replica sets are consistent whilst allowing controlled scalability.

Local Deployment

Having local systems has been the prominent method for deploying integration software since before banks began building display systems in datacentres. By deploying servers, mainframes and computers on site - sometimes known as on-premise or on-premises - enterprise are able to own the software rather than leasing. This gives a greater control for the upgrading, updating and management of the servers.

By deploying in an enterprise site, it is possible to meet in house requirements such as security needs like ensured traffic isolation. It also allows traffic throughput, memory etc to be managed internally without fear of poor practices by other users of the system causing system level issues.

Availability and disaster recovery can feel better handled in a local deployment because distance, distribution and sizing are controlled by the enterprise. Some systems require data centres in a specific radius of each other to avoid disaster outages but to still allow fast sending of data e.g. for quorum or replication requirements.

Unlike cloud deployments however, on site systems do not necessarily have high scalability because servers can take a long time to provision, configure and setup whilst needing both the hardware and the software to be maintained and managed by in house team with the relevant skill sets.

Microservices

Through SOA - Web services had grown massively, but so too had mobile and web applications and a need for stateful smart services that would keep data with logic in an object-orientated programming structure.

In 2005, Dr. Peter Rodgers used the term “Micro-Web-Services” at a cloud computing conference. In 2011, a conference of architects coined the term “microservices” to describe the new architectural style, with the method name formalised in 2012. It was later further discussed by James Lewis and Martin Fowler in an article on the subject. By 2015, Sam Newman had written a book describing how to build microservices [82], [83], [84].

Microservices are different to web services in that web services are the exposure of application functionality to provide greater integration of applications and systems, whereas microservices are a collection of modules that together form a larger application. They run independently and can be message or event driven API’s - although not all API’s are microservices.

The ideology of microservices is that the breakdown of services means that developers can quickly make changes to a single service without the need for the whole application to be taken down or changed, a concept fuelled by the advancement of “Continuous Delivery” a term popularized in 2010 by Messrs Humble and Farley [85].

Microservices are the latest step in an ever changing, ever improving integration world. But microservices are an architectural concept which has been the focus of some fierce debate within the integration community, as it echoes the architectural style of Service-Orientated Architecture [86].

This level of service integration sees runtime services configured to the needs of a business process. Either through pre-configured services stored on a repository which can be accessed and queried by the required services [92], [101], [102].

Whilst the topic of SOA is contentious and broad in its scope, it is a vital way in which business and enterprise plan and invoke their integration needs and will continue to do so alongside and in harmony with microservices.

Where Microservices can be very useful is when they replace existing tightly coupled applications. Microservices should complete a single process and be lightweight. The idea is that microservices can be quickly deployed to replace or extend an existing microservice application. This might be because of increased traffic on the system, to allow for a new version of the application to be introduced, or to allow new functionality to be introduced.

The deployment of multiple microservices can quickly become enormous if entire integration environments are deployed in the same microservice. If, for example, an application was used to get an advertisement collection from a system of record for each new user to a page and also included the logic and functionality to process payments for sales made on the same site, in the event of a new deal being launched that brought large numbers of customers to the site, new deployments of the microservice would also include all the infrastructure, libraries and 'baggage' of the payment system as well.

Microservices are intended to be designed so that they are decoupled and self-contained with the ability to be deployed using multiple technologies. This might be Ruby on Rails, JavaScript, C, Python or any other language or underlying system. This makes it easy to deploy, reuse and remove from a system whilst being comfortable for developers within the enterprise to implement and update.

This brings with it its own risks, in that if an enterprise creates or allows the creation of several different coding languages across the full system stack, there comes with it, the need to have developers of lots of different language types and can mean bottle necks for development if there is a lack of skills.

Having each service as a contained system allows for better testing of individual points of service and integration. Unit testing and component testing allows issues to be caught early and at specific points of the system. However, there is a risk that test plans can become large and cumbersome in themselves, with lots of management of the testing suites used to keep the microservices working. It is also difficult to ensure that tests are written at all, which is a common issue alongside lack of documentation, especially as development cycles have shorter time spans.

One of the more contentious logical issues in a microservice is the requirement for stateful services against the difficulty it brings to the principle of having a throwaway instance that

can be scaled both up and down. Stateful services cannot easily be scaled down and would need its own separate tools to do so without losing integral data.

Asynchronous messaging raises a similar concern with or without state, in that it is difficult to understand when a message needs to be reverted or rolled back, see ACID.

Microservices separate each task ran by the enterprise to a single callable process in order to allow each component to scale up easier, be easier managed and to not affect other services when altered.

12 Factor Applications

One of the ideologies behind the creation of Microservices is that applications should be portable, resilient and easily deployed to cloud environments. Developers at Heroku drafted a 12-factor methodology for running applications as a service with the view of making them automatable, portable, cloud deployable, minimally divergent and scalable.

The 12-factors were written on https://12factor.net by Adam Wiggins, a co-founder of Heroku - a cloud platform as a service which was established in 2007 - as a way of describing the common pitfalls that are encountered when applications are deployed onto the cloud.

The twelve factors are:

1. One Code Base
2. Explicitly declared and isolated external dependencies
3. Environment specific deployment configuration
4. Backing Services such as databases are treated identically as attached and replaceable resources
5. Build, Release and Run are used to separate distinct stages of development and deployment to avoid knock on effects or overlap between them
6. Runs as one or more stateless processes with no shared resources
7. Self-contained with a service endpoint on a well-defined dynamic hostname and port
8. Scaling is completed at a horizontal level via additional process instances
9. Instances are disposable with minimal start up, graceful shutdown and toleration for abrupt process termination
10. Designed for continuous development and deployment with minimal differences between the application in Development environment and Production
11. Logs should be used as event streams with the hosting environment handling processing and routing
12. One-off administration scripts are kept with the application to ensure they run with the same environment as the application itself

Benefits:

The primary positive of a 12-factor compliant application is the ability to scale up and scale down during busy and quiet periods. This relates directly to the main container discussion much earlier in this document whereby on a hot summer day, sales of fans might increase seeing increased demand on a shops online traffic.

With well-produced cloud applications, the creation and deletion of application instances is much easier and poses little to no risk for the application owner or the customers. In an ideal world where no state is held on applications, there would be no adverse effects, such as loss of an order when an instance is deleted because of lower traffic demands.

This improved resiliency gives an enterprise peace of mind in increasing and reducing the cost of running their systems throughout the year without negative financial or reputational impact.

With a 12-factor application that has close alignment between development environments and production environments, an efficient 'continuous delivery' model can be adopted. Continuous Delivery (CD) relates to short releases of product functionality and code with regular updates made to a system. This model was an expansion on the continuous integration technique which was first used in an experimental software development environment which looked to integrate strongly connected modules in a hierarchical way down to the least connected modules.

In this way, change can be better coordinated when software is evolving and being maintained [154]. One example of this at work is during an enhancement of an existing system whereby a developer wishes to get code into the route-to-live. Their changes are made locally - and tested against a set of unit tests - before being pushed to a code repository where an automated deployment pipeline implements the changes onto the system using a series of scripts. When followed with further tests the Development Operations (DevOps) pipeline can push the changes right the way through to the live environment.

By allowing applications to follow CD releases, functionality improvements and fixes can be made to a system with fewer issues raised than in a big release update that happens disparately throughout the year. This is because issues found are small and can be quickly fixed with team focus in relation to the upgrade task.

12-factor applications also see benefits from being more standardised and allows easy and secure repeatability as code is pulled from a code source repository. This reduces environment disparity and is also easier to pick up for new members to the team as they are familiar with what an application should look like, what it should and should not do and means that changes made in lower environments will be very similar to the effects in Production.

Finally, 12-factor applications have improved security in that credentials are no longer stored in the code base itself - or if it is it can be locked down to a subset of users with the correct permissions to access. Without credentials hard coded into the application, it is much more difficult for wrongful or malicious use [150].

Disadvantages:

What can often be the downfall when following the 12-factor approach is that many use the factors as hard lines in application creation, and not, as with all models and standards, a set of guidelines to be used and aligned to as closely as possible.

Standardisation of applications and systems is a valid and correct approach to take but does not fit for all enterprises. An example would be that whilst the applications should not hold state, this becomes very different for systems where state absolutely must be held. If a payment system that was adding or removing money from an account was to remove money before an order had been confirmed, the customer would be charged for a service or product not received.

This concern is amplified within banking systems where payments are made in and out at every transaction. This is also true for highly critical systems such as train and air control where live tracking ensures that passenger and crew safety is not put at risk.

Additionally, standardisation of systems can be hard to enforce and to be sure where the standard is no longer being followed. Versioning as an example is largely unregulated across enterprise systems, until the system fails because a new piece of code was given the same version number and caused confusion in the route to live. Enterprise need to be aware and weary that standardisation requires management.

Another key issue concerns the scalability of a system. The first in that stateful systems cannot easily be scaled down. The second, is that the ability to scale does not mean that the underlying system can scale, as the underlying infrastructure whether that be on-premise or in the cloud, still requires appropriate sizing.

Nor is the discussion and factors raised by Heroku specifically ‘new’ but are old problems that have been in the industry for a long time. What gave success to the 12-factor app is that the write-up described the issues, the solutions and case studies for success in a consumable and repeatable format.

Enterprise should look to follow the 12-factor approach as closely as possible with thought placed on whether the rules are applicable to the current system when put into context of the risk should the system fail.

Cattle Vs Pets

With growing use of application in cloud environments, and specifically in the context of scaling up and scaling down the number of servers in an environment for a given application or server, there was a growing need to be able to effectively explain how containerised platforms differed fundamentally from standalone fixed servers.

In 2011 and 2012, Randy Bias of the Open Stack foundation took a presentation by Bill Baker on scaling up or scaling out of SQL servers and expanded the concept to be applied in cloud applications [155].

In the analogy, “Pets” are treasured as something with a sort of emotional attachment. Try asking a system owner if you can rebuild their environment in a traditional environment to see the affect generated. The servers are manually built, managed and maintained (“hand-fed”) by the system owners and are indispensable to the system as a whole. There is a reliance on pets to have high availability and to alleviate the risk of server outages by having running - or ready to run - copies of the servers to take over in the event of failure.

The problem is that this is expensive both in terms of time and cost. In a system where a single server can handle the full throughput of transactions per second (TPS) there is a minimum requirement to have at least two servers in case the first one fails. For further resilience a second site must also exist with an additional two servers in the event that there is a failure in the first site, thus totalling four servers for each single server that can handle all the TPS.

Conversely, “cattle” refers to multiple arrays of servers where the failure of any single server or even collection of servers are replaceable. In the cattle system, automation occurs to ensure any server that fails is quickly replaced with the routing and handling of traffic managed without human intervention. In the analogy, the cattle are dispensable and easily deleted without the loss of the systems functionality.

By breaking up existing systems into multiple separate and decoupled applications, services and runtimes - enterprise and organisations are able to evolve existing ‘pet’ systems into ‘cattle’ systems.

Typically, pets will have resource efficiency, multiple interdependencies, designed for longevity and centralisation with a focus on maintaining what exists. Cattle systems will be disposable, will be isolated, have agility whilst utilising decomposition and will be elastically scalable.

A typical misconception when attempting this is that by moving everything that already exists into containers will automatically create a ‘cattle’ approach, however, in reality there are extra steps to be taken to ensure that scalability, resilience and agility [156].

This might be that the centralisation that has typically occurred no longer fits for the new disposable runtimes and that instead each runtime will need to be compact and decoupled. An example might be in API Hubs, service registries or application servers, where an entire organisation worth of services, applications and APIs are bundled and managed as one. For the 'cattle' approach to be truly taken advantage of; small collections of runtimes will be needed to allow systems to be scaled and composed correctly by the container management layer.

With the consideration of availability, the resiliency of a specific service and even scalability, the decoupling of runtimes affords greater isolation of resources and shared resources. This might be threads, memory, CPU, hard drive, temporary disk space or even just concurrent connections and workload management.

Each organisation, team and project will need to make informed and sensible decisions made around the level of granularity that is required for the systems being created. Much in the same way that CAP theorem has trade-offs, so too does the cost of splitting runtimes against the benefit gained. Purists might find that when it comes to the financial budget review for the following year, the proposal for additional resources to isolate each runtime will not be fruitful especially in a system that does and has worked for many years [157].

There are further considerations such as how are each of the runtimes logged, how do service credentials get shared easily amongst the different runtimes and again the discussion on state becomes prominent despite all microservice approaches being compatible with retaining state providing there is reliable persistent storage.

In summary, Microservices allow the creation of portable, decoupled services that can be deployed across many products, software's and hardware's. They offer enterprise's better version control and testing as well as allowing better management of independent and scalable applications.

Interface Characteristics

The sorts of architectural choices that business will need to make, will focus on the interface characteristics and the questions posed in concert with them [147]. Interface characteristics describe how the interface will work functionally, how interaction is completed and the usual non-functional requirements including reliability and availability.

The characteristics described below apply to all architectural types including Hub-and-spoke, AIA, ESB's etc and are important considerations when designing and architecting interfaces.

Functional:
Functional characteristics will need to describe what sort of object is being used and the functionality it will entail. For example, a SEBO might be chosen as only using RESTful GET

operations. Which of the Create, Read, Update and Delete (CRUD) interactions will the business object use when connecting to a data store and will the Request/Response object be stateful?

Technical:
Technical considerations will include the transport to be used; HTTP, TCP, MQ. The protocol and wrappers used to send data e.g. HTTP(S) or SOAP and what format will the data be in. Text or Binary? How will dates be displayed? YYYYMMDD or DDMMYYYY for example? Each will specify acceptable forms of input for communication.

Interaction Type:
At this stage of the architectural discussion the subject become more difficult to answer with multiple choices having consequences on the entire design, budget and sizing of any software solution. Will messages be bound to 'Request-Response' in that they send a request and must deal with a response or is 'Fire-Forget' better suited whereby requests are made, and no response is necessary to continue processing other data? Will asynchronous be used or is it preferred to use 'Thread-Blocking' in order to wait for a response before doing any other calls?

How big should each message be, and should messages be dealt with in batch i.e. process salary payments all in one go at the end of the month, or should each new message be handled individually?

Performance:
How do we measure response times and what is an acceptable response time? How much throughout can be handled by the system and at what volume? Are requests forced to occur one by one on some systems or can there be an element of concurrency?

Integrity:
To ensure integrity of the system, validation rules may be needed at each business object message exchange. Are there any mandatory fields? Has the correct format been used? Is transactionality important - where all messages sent in order and at the same time or not at all? What about state? Should a stateless protocol not reliant on the server retaining session information be created, or should the protocol be stateful and require session information be stored?

What is the event sequence? Does your bank need to request that you confirm an online payment and wait for your response or can it process the payment predicting you will confirm? More importantly, can duplicates be abided anywhere on the system? What level of Idempotence is acceptable?

Security:
Will users have an identity? Will the system? Will both or neither? What authorization is

required to complete a transaction and how is this checked? Is data being kept private and who owns data at each stage of the architecture?

Reliability:
Should the system be always available? Can there be any downtime? Should consumers be assured of the delivery of their message and subsequent processing of the request?

Error Handling:
How should errors be communicated? Where are they stored and dealt with? Whose responsibility is it to debug and fix an error? The provider, the consumer or the customer.

Qualities of Service

There are a series of factors that make up the general quality of a service in terms of what the system owner can expect from the system. This ranges from the assurances of delivery, the performance, handling of errors and general ordering of messages.

ACID

Transactional messaging - a series of operations performed in totality or not at all - must be able to track, rollback and retry as part of the message delivery. These transactions are defined as Atomic, Consistent, Isolated and Durable (ACID).

Transaction may fall into one of three main categories which are;

- Unprotected - In the event that a transaction fails, is aborted or is malformed there is no need to try the transaction again. This might include the download of program from video sharing platform to your phone
- Protected - Transactions must be re-attempted or rolled back if the transaction fails, breaks or needs to be aborted. This includes your typical messaging examples e.g. an update to a backend database for a new customer at a bank
- Real - Once the action is done, it cannot be undone. This would include the initial deposit payment of a home-purchaser, or the delivery of medicine to a hospital

Each of these transactions can have one of two outcomes - committed or aborted [151]. It is worth noting that there will obviously be some example where the categories do not fit.

Jim Gray, a computer scientist from the USA who worked throughout his life at Microsoft, IBM and Tandem Computers was a system engineer who gained an ACM Turing Award in 1998 for his work on the concepts and implementation of reliable transaction processing through the ACD model [152].

Atomicity

Atomicity is the determination that a transaction happens or that it does not happen, and that all parts of the process are bound to the same service and execution or that none are. The only way the transaction can succeed is if all the component operations also succeed.

In a wider application and service context, atomicity describes when an operation or set of operations appear to occur without a break or pause. This means that the operation(s) are not interrupted by other processes, threads or signals. It is, however, possible for the operation(s) to not complete instantaneously but it is integral that it appears to be from the rest of the system. The aim is that the operation(s) are free from any side effects that might occur if other processes or operations have the ability to interfere with the transaction.

For read only transactions, atomicity is easy to achieve in that if the request to read the information fails then a retry can be attempted. The file is either read, or it is not read. With more complex operations such as the processing of a payment or writing to a database, the system MUST be able to rollback changes that have been made so that the system is left in the same state it was in when the operation was primarily attempted.

Consistency

Transactions retaining consistency might seem a fairly basic concept, in that data structures and the data committed must satisfy the application. A common example would be where a number being placed into a database must satisfy the column value type. It could not place the number into a string only column because this would be inconsistent with the application.

This can be applied in general coding practices too, in that a string cannot be replaced by a number, though it might be replaced by a string copy of a number e.g. “2” instead of 2. In a database system the expense of checking each column of each row for consistency both before and after the transaction is too great to justify. Instead, checks can be made against the commit itself.

Similarly, this is true for testing systems in application and service code. The state of each entity of the application can be tested in its entirety for irregularities to confirm consistency but the sheer expense could be staggering. One example might be for an instance of MQ which is used for a transaction, checking the consistency after a transaction of every Queue in every Queue Manager would see lots of organisations validating tens of thousands of queues for every transaction.

Validating input to an application serves to provide the cheapest technique in that each variation of input data can be pre-tested against the applications input validation module for handling of inconsistencies. In this way live data can be assumed to have a consistent effect on the applications used in the transactions.

Isolation

By controlling concurrency, applications give the appearance of operations being detached from outside systems. Apparent isolation means that processes and threads that are not part of the transaction do not see or are aware of the changes in state that may not be fully committed, such as in the event of a transaction failure.

Concurrency controls such as managed locking mechanisms mean that operations can block the updating of a record by other applications whilst the transaction is interacting with the record. There are two main characteristic types of locking known as ‘optimistic’ and ‘pessimistic’ whereby a system or record is either locked during reading of a record, or it is not.

Pessimistic locking isolates the system of record for the application’s use only which is a much safer way of ensuring integrity of records but comes with the additional risk of locking the system entirely to all applications in a process called ‘dead-lock’. This has the effect of complicating system architecture.

Durability

Successful transactions must be persisted in the system of record - hence the use of the word durable. Gray refers to this as the transaction becoming part of the ‘reality’ of the system and thus, cannot be forgotten.

When the system has acknowledged the success of the transaction, the onus is on the system to be able to handle any future failures from the environment, user or the hardware, such as if the hardware reboots, the transaction MUST be retained.

For all future changes that need to be made to the system, an additional transaction is required as the systems new reality has the transaction included. There may be an ability for the system to roll back the transaction from a user experience point of view e.g. email recall which recalls a sent email from a user’s experience but still requires a secondary transaction to remove the record from the system under the covers.

These four principles have been a foundation in architecture design of system transactions for many years, even though the original paper by Mr Gray did not discuss isolation. In a co-authored book from 1993 with Andreas Reuter, the concepts of ACID are described in further detail including isolation which he adds “…(And, of course, isolation is also required to make the acronym pronounceable.)” [153].

ACID is not without its own challenges. From the initial concept of ACD, the number of transactions per second expected in modern systems have increased exponentially, for example, in the context of the e-commerce revolution of the 90s.

Increased transactions lead to the need for concurrency which adds a greater level of complexity to each of four principles especially when all transactions must be dealt with as

if they are serial. It is concurrency, resilience and availability that microservices are looking to improve.

Message Ordering

When request transactions are sent to a messaging application, it is important that the order in which they are sent is honoured. This holds especially true for asynchronous calls which arrive on the application as and when requests are processed.

Types of transactions which might require correct ordering include ticket reservation, payment services, event logging and many more. These systems may run based on a First In, First Out (FIFO) principle and as such the message applications need to be handle the ordering of events. A practical example would be that a reservation confirmation should precede a reservation cancellation.

By explicitly setting message ordering, the processing and processing rules defined by the application can lose some of their complexity but might come at a cost of scalability (as messages will be in effect synchronous). It is also more difficult to manage the ordering of messages across a multiple server system as each message must be tracked and its status synchronised across the servers.

There are some disruptions to typical message flows that should be considered as design is taken into consideration, such as pattern types being used e.g. fire-and-forget, error processing and handling, and message priority which can lead to messages getting out of order as more ‘important’ messages arrive on the system.

In pub-sub systems, messages are not necessarily delivered in order to the subscribers of the messages, making this type of messaging a more difficult implementation to use when wanting to ensure the order in which messages are received.

Where multiple publishers or multiple subscribers are sending messages on the same topic the system can have greater control of ordering by allowing the publishers and the subscribers to coordinate order assigning, as part of the sending of the message. Additionally, messages themselves can hold data which describes their order, usually through a transaction storage system.

Error handling also raises a requirement on ordering because rolling back transactions and resolving in doubt transactions may be handled in a way that the message is technically ‘sent’ but is awaiting confirmation from the sender in order to continue message processing.

It is also important to consider exception scenarios when creating message flows, as strict ordering can cause issues to the quality of the service if there is significant time spent resolving issues, such as the destination being unavailable or unable to handle the message.

Transaction Handling

In an ideal world, transactions complete successfully without duplicate records being sent and without issues on the network or within applications. However, this is not always possible and so, platform resilience should be planned when designing integration patterns.

Retry

There are many reasons why a transaction might not arrive at its destination target; network connectivity issue, message queue is full, channel is at its max length, load is too high or one of the servers in a cluster was temporarily down. Integration patterns should allow for a retry process of which there are three notable options available;

Message Retry - in most applications there is an ability to set a retry limit usually alongside a timeout value. The sending application will attempt the first message send and can check for failure notices. Depending on the error type, this can lead to another attempt to send or to analyse the period of time that has elapsed.

Setting timeouts is important primarily because it ensures a single transaction does not block other services entering the system, which is particularly prevalent in synchronous message flows - especially where the message receiver does not reject for whatever reason.

An alternative to out of the box message retries is to define custom retry processes. This gives the ability to set different process flows for different types of failures. For example, if message sends are consistently receiving server errors from a specific endpoint, an alert system can be invoked to trigger a set of rules e.g. restart of the server or a log dump to a relevant team.

Return to sender - Another option for failing messages is to return the message back to the sender, moving the onus from the application to the sender to ensure the delivery of the message they have sent. This is usually determined by the sender of the message as some form of description in metadata, headers or within the message itself.

An example of how this works day to day is the postal system where a letter is written (message) with an address on the envelope (destination), on the front of the letter is the return to sender address (return description) and is placed into the post box (channel) and taken to the post office (application). At the post office a sorting machine doesn't find the address in its database (error) so says to return to sender.

This is beneficial both for the sender and the receiver because the sender does not have to keep the message for later transaction and the receiver knows the message has been returned to them which is important for once and once only transactions.

Message storage - If a message cannot be sent or returned, the message can be stored for later processing on a dedicated channel or store within most applications. The channel can

later be used to complete a batch process that will handle the ‘dead’ messages based on the failure type.

A batch process - originally used in factories to describe completing a lot of the same process all at once - such as the manufacture of a single part of a wooden horse. Since 1964 and the release of the IBM System/360 computerized batch processing refers to running a script against one or more programs which might include compilation and deployment.

The storage of ‘dead’ messages allows messages to not block existing channels and message queues meaning a failing backend will not negatively impact the whole system.

Idempotency

The introduction of retry processes for failing messages can lead to problems when the result of the message being received has a direct impact on the end user of the system. That is because the retry may be improperly invoked for a supposed failing message but that was actually received already.

A good example would be a cash withdrawal message from an ATM to the bank, if the same message was received twice the users balance will be down by double yet they would have only received half of that balance as cash.

To alleviate this risk proper transaction handling is put into place to ensure the system is idempotent meaning that if the message is sent twice the duplication can be handled without the affects being applied more times than intended.

In mathematics the term means a function that has the same result when applied to itself. Because the function is applied at the receiver level, idempotency should be used by the message receiver.

One of the simplest ways to achieve this is for the request made by the sender to include a unique identifier (ID) that can be tracked for usage by the receiver e.g. I have already processed a message with this ID, do not repeat the transaction processing at the receiver.

Another method might be to send a cached success result as a response for any errors that are seen to have caused the successful processing of the message on the server’s backend systems but has failed somewhere in the response.

In some cases, idempotency might be set on the channel itself as a deliberate fail safe against chosen architectural practices such as a lowered quality of service - to improve performance - or because the sender cannot be sure it has successfully sent the message to the receiver the first time.

Duplication

Message duplication might be handled separately from an idempotent service by only acknowledging that a message has been sent in the event that the channel service has received a message successfully.

By having strict checks on the response output from the receiver it is possible to reduce duplication possibility. This can also be true when the channel or network between the channel and the receiver is disrupted through synchronisation of state between the sending and receiving application.

If no acknowledgment is received by the sender than the message will not be deleted but a request for acknowledgement of the state of the transaction may be sent instead by confirming if an equivalent transaction has been completed by the receiver already. This might be either through a unique identifier or through transactional input that allows transaction identification. In a ticket reservation system this might be a correlation of name against seat number and card details.

Compensation

Compensation occurs in transactions that have not achieved successful acidity or where a process is spread across multiple transactions. In this case compensation is required to allow the rollback of the action taken to ensure the integrity of the system is retained.

In essence the act of compensation is used to revert a system back to its original state by placing a secondary transaction that understands the divergence between the ‘then’ and ‘now’ state. The full process is handled by three main parts; logic, activity, faults whereby logic describes what should be implemented, activity triggers the compensation logic to begin and the fault is used to handle any issues that may arise.

The gap bridged by the compensation method comes into place mainly where two-phase commit (2PC) does not sufficiently ensure that all transactions will be completed, or else sub transaction failure will result in no loss to system integrity. The aim is to get the system back into a consistent state without aborting all the levels of the transactions.

Error Processing

It is important for message errors that may occur to be correctly handled for processing either by the sender (in the form of a response), the channel (to determine what to do with the message e.g. retry or return) or the receiver.

By setting the error code and error message it is possible to describe the type of error to the various stakeholders in the flow and thus each error can be handled by the sender, channel and the receiver.

It is typical for specifications, schemas and error processing guidelines to be determined at industry levels through industry standards which may or may not be regulatory required, or through enterprise determination.

One example would be the use of Swagger definition for REST API YAML descriptions which sets schema standards for message parsing. One example is Open Banking - a banking API exposure regulatory project - which relies on swagger specification which mandates regulatory responses to specific error scenarios which must be given by the API provider.

This pattern of determining what should and shouldn't be provided as errors are seen throughout the integration industry often in RFC's for specific new technology types.

Structured Data

Data is integral to the integration journey; it provides the detail of messages sent and received. It provides the information required to complete some form of task such as providing a customer id number when logging into your online banking.

Data can either come in an unstructured format or a structured format. For unstructured data, information is written in free form with no set direction or method, think; notes in a lecture, post it notes with jotted information or even in the conversation with a chatbot.

Structured data generally refers to data that is organised, maybe into headings, a set style or a table. In one example a database is the perfect example of structured data, it has headings and each new piece of information inserts data into the relevant columns as new rows.

Structuring data allows enterprise to configure integration products and patterns to quickly consume and produce information in messages. An example might be;

Unstructured data:

"I have two accounts - a saving account and a current account. The first has £11, the other is overdrawn by £12."

Structured Data:

Account Type	**Balance**
Savings	11.00
Current	-12.00

The second is a smaller message, the relevant information is already ordered, and no natural language processing needs to be completed on the incoming text.

CSV

Comma separated value consists of a document which contains 'columns' of data. Each column is separated with the use of a 'comma' in order to separate each value. If a column does not have an entry the data would be shown with a double comma. Let's take a form that expects a first name, middle name and last name, a person without a middle name would leave the middle field blank e.g. Ryan,,Read as oppose to Aiden,John,Gallagher.

Comma Separated Value was once an easy way to separate between punch card slots of data, such as how the IBM Fortran OS/360 did in 1972. By 1978 a freer form input output had been outlined by Fortran 77. CSV requires each record to have an identical set of entities or "fields" which historically made it more reliable when holes were punched in punch cards slightly out of column [76], [77].

XML

Extensible Markup Language (XML) formats documents that are both machine readable and human readable. This is important because whilst the data is being processed by the machine, the human is able to understand, debug and validate information. It also provides a human language for software validation of incoming text i.e. the value for <name> should be Bill. This is easy to check by looking for: <name>Bill</name> and is simple at all levels of an enterprise both technical to non-technical roles.

The history of XML began at IBM when Goldfarb, Mosher and Lorie invented Standard Generalised Markup Language (SGML) in the 1970's [55]. SGML in itself is not a Markup Language but instead is a language to specify other mark-up languages. What followed was the modern webpage presentation technology which was developed by Tim Berners-Lee in the late 1980s – HyperText Markup Language (HTML).

In reality HTML was not well used for data storage and interchange but its predecessor SGML was seen as too difficult to implement on its own [56]. In 1996 the world wide web consortium (w3C) had enlisted developers that included; Jon Bosak, Tim Bray, C. M. Sperberg-McQueen and James Clark to produce a slimmer, more usable version of SGML which could be reused and consumed unlike the 150-page original SGML specification which accounted for masses of irrelevant and complex procedures [57].

XML quickly became the bridge between SGML and HTML becoming widely used by the computing industry and can be found in a host of processes and transactions.

JSON

JavaScript Object Notation was "discovered" by Douglas Crockford though he claims that java array literals had been used in Netscape as far back as 1996. Though Mr Crockford made the specification and a website.

The first JSON message was sent in 2001 when a server responded with:

{to: “session”, do: “test”, text: “Hello World”}

Which initially failed due to “do” being a reserved word. This led to all keys needing to be quoted making it simpler and easier to use for other systems such as python [78].

JSON is a self-describing, hierarchical structured simple text which is both easy to use and roughly two thirds more compact than its XML counterpart [79]. Although its name contains “JavaScript” it is actually language independent.

RSS

Rich Site Summary is also known as really simple syndication and is used in *aggregation* to collect continuous data from the world wide web. RSS was initially pioneered by Netscape in 1999 before RSS feeds were offered as a service on the New York Times website in 2002 [52].

RSS was important because it allowed information to be disseminated to an interested party by providing updates either from a page or a topic. For example, sites were able to use RSS to bring together news articles much in the same way a user of the BBC News application in 2019 might select topics of interest. Conversely, users are able to provide their own RSS feed to an RSS feed directory which provide RSS subscribers to gather information from various data sources.

The RSS feed was - and in some environments - still is available to use on websites, but more commonly information is now portrayed via social media by ‘following’ someone or something of interest such as ‘liking’ a favourite band on Facebook, ‘following’ them on Twitter or searching their ‘hashtag’ on Instagram or by selecting interests on Pinterest or StumbleUpon.

Fixed Width / Format

Fixed width applies to formatting data in a way that sets a specified width of data per field, with both the width and the character used to pad out any empty spaces configurable by the data owner. An example might be a fixed width of 10 characters with ‘#’ used to pad the remaining space, where the fields relate to first name, middle name and last name.

Aiden#####John######Gallagher#

Fixed width might also include a header at the top of the record to be used to identify the field names and types with each new record separated by a new line character, although an object model can be used instead.

firstname#middlenamelastname##
Aiden#####John######Gallagher#
Andrew###############Garratt###

Most commonly used in flat-file databases, using fixed width formatting instead of delimiters can lead to an improvement in processing time as the processing demands are already outlined rather than having to check each character for the chosen delimited value.

Protocol Buffers

In early 2001, developers at Google were looking for a way to serialize data in order to communicate across machines or to store it. In creating the protocol buffer - sometimes known as ProtoBufs - Google hoped to create a smaller, faster and simpler version of XML in order to allow defined structures to be used to generate reusable source code for passing data to and from data streams in different languages such as; C, Go, Java and Python.

Users of the open source code - released officially as v2.0 after some rewrite of the internal code – began by creating a 'Message' record of small logical information containing field names and value types. Fields might be; required, optional or repeated.

The message is then compiled using a language specific compiler which creates objects for the coding language to be used in order to serialize, populate and retrieve protocol buffer messages using a determined schema [183], [184].

ProtoBufs are fast processing in comparison to some of their modern competitors such as JSON or XML simply because of the dependency on string encoding/decoding and handling of human readable elements like whitespace on the text-based data. However, this advantage relies on well written code and as such can have similar performance outcomes if text-based data is written better.

ProtoBufs use binary serialization which makes the payload smaller, math simpler and uses integer lookups that results in making it faster. This explains the claim by Google of a smaller, simpler, faster method than using XML. However, the trade-off is that the protocol buffer is not human readable and therefore can be hard to debug and requires the use of a schema alongside any data models used.

Conclusion

Since the 1950s and 1960s, enterprise have evolved and adapted to perform the functionality that government entities, customers and organisations have needed. With the goal of providing ever quicker reliable and scalable systems there has been a shift from isolated islands of communications through to containerised self-built island networks that are managed, created and deleted as need demands.

Throughout this book we have seen the foundations for which applications and services are built upon; the architectural styles, how code has had to change, individual software with the problems they have looked to solve and the considerations for each when implementing in real life, outside the realms of these pages.

It is important to understand the hurdles that have been overcome to create the systems that are seen and used today. It is also important to contextualise each of the parts into its wider setting by asking the question of 'What is this solution solving?' and 'What are the real-life impacts?'. When each of the adaptions are put into context it greatly benefits the reader to see how little by little, the industry has moved from code cards through to client server runtimes that can be used to create impactful, every day person facing applications, in minutes.

Whilst past innovation had mainly been through government and university collectives the key to the success was an availability of information and talent, as well as access to the resources required to take an idea and turn it into something tangible. Through improvements across the technology industry - of which integration has been pivotal - information is much more widely available as are the tools needed to create and test practical solutions to problems faced every day.

The ability to improve customer interactions, business to business communication and applications for the public is now well within the reach of "bedroom developers" with little more than a laptop and internet access.

Whilst there are huge benefits to integration in its current form, the cycle of all products show that everything can be improved, often with small (seemingly obvious) changes that surpass all expectations. Think of the smartphone which holds a calendar, camera, calling, messaging, news gathering and game facility that allow access to all topics and subjects at the click of a few buttons.

In writing this, we have considered how the integration landscape might change in the coming years. A highlight of some of the difficulties that will need to be solved in the next decade will be;

- Integration Databases – shared databases of multiple applications - require close coupling making expansion and evolution of the applications (and possibly the

database) difficult to implement.

- Scalability will remain an issue with container and container management systems needing to become more reliable, easier to consume and with a lower barrier to entry.

- The industry as a whole will require dramatic learning and upskilling techniques as individuals are expected to be able to deliver much broader scopes of integration architectures, techniques and software's.

- The availability of APIs - both public and private - is and continues to be a large concern. With fast paced development, enhancement and change it is and will continue to be difficult for reliant companies or users to capitalise on the value gained.

  One example is that of twitter when it enforced third party twitter applications with more stringent API rules [103]. The same happened to google reader when the company reduced and then pulled the API due to a fall in popularity a move that led to some hesitance in adopting Google Keep from weary developers [104], [105].

- With the expansion of the microservices architecture the types of services that are made, will become developed detached from the developer. It is then in the hands of the enterprise to manage the service deployment, scaling and resilience. In the event that components aren't well composed, the complexity moves to connecting the components which is harder to control.

- The management and orchestration of large enterprise solutions both in terms of governance and deployments will continue to be encompassed by infrastructure and their teams' dependencies. Simply allowing isolation of teams within a platform will not suffice as the wider platform considerations will need to be taken into consideration.

The potential issues with integration are substantial and remain a concern for enterprise, these range from governance issues to security implementation through to HA implementations not being sophisticated enough in that they are costly and difficult to maintain and produce.

Each innovation and each improvement bring new problems to be solved by the next generation of integration pioneers, whether it is moving data faster, making systems more reliable or storing data securely.

A common theme amongst all the changes and adaptations that have been made is that they are built on something that already exists, pulling together ideas and functions from

across the integration landscape to make small changes. Each new idea is a refinement on the ideas implemented before. Nothing is ever really new.

“The thing that hath been, it is that which shall be; and that which is done is that which shall be done: and there is no new thing under the sun.” – Ecclesiastes

References

1. Peter Bernus and L. Nemes (ed.). (1995). *Modelling and Methodologies for Enterprise Integration: Proceedings of the IFIP TC5 Working Conference on Models and Methodologies for Enterprise Integration, Queensland, Australia, November 1995*. Chapman & Hall. ISBN 0-412-75630-7.
2. Jim Fenner. Enterprise Application Integration Techniques. *Lecture delivered in Advanced Software Engineering, Unit Code: 3C05.* 03 May 2003. University College London. [Online] Available from: http://www0.cs.ucl.ac.uk/staff/ucacwxe/lectures/3C05-02-03/aswe21.pdf [Accessed 02 February 2017]
3. The Open Group. The Open Group Service Integration Maturity Model (OSIMM) Version 2. The Open Group Website, 2016. [Online] Available from: http://www.opengroup.org/soa/source-book/osimmv2/index.htm [Accessed 02 February 2017].
4. Gregor Hohpe and Bobby Woolf. (2004). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Boston, USA.* Addison Wesley. ISBN 0321200683.
5. Wesley Fenlon. Blame your internet latency on the speed of light. *Tested website*. 18 March 2013. [Online] Available from: http://www.tested.com/tech/web/454189-blame-your-latency-speed-light/ [Accessed 18 February 2017]
6. TeleGeography. Submarine Cable Map. *Submarine cable map website*. May 2017. [Online] Available from: http://www.submarinecablemap.com [Accessed: 18 February 2017]
7. A. Bhushan. A File Transfer Protocol. *Internet Request For Comments (RFC) 114, MIT Project, 16 April 1971* [Online] Available from https://tools.ietf.org/html/rfc114 [Accessed 04 February 2017]
8. J. Postel. File Transfer Protocol. *Internet Request For Comments (RFC) 765, ISI, June 1980* [Online] Available from: https://tools.ietf.org/html/rfc765 [Accessed 04 February 2017]
9. J. Postel and J. Reynolds. File Transfer Protocol (FTP). *Internet Request For Comments (RFC) 959, ISI, October 1985* [Online] Available from: https://tools.ietf.org/html/rfc959 [Accessed 04 February 2017]
10. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.88.7505&rep=rep1&type=pdf [10]
11. Barry M Leiner, Vinton G Cerf, David D Clark (et al). Brief History of the Internet. *Internet Society, October 2012* [Online] Available from: http://www.internetsociety.org/internet/what-internet/history-internet/brief-history-internet [Accessed 05 February 2017]
12. IBM Corporation. History of CICS. *IBM website 11 December 2015* [Online] Available from: https://www-01.ibm.com/support/docview.wss?uid=swg21025234 [Accessed 05 February 2017]
13. Diego Perini. What is a database? *BBC website 02 October 2010* [Online] Available from: http://www.bbc.co.uk/guides/z8yk87h [Accessed 06 February 2017]
14. Susan de Sousa. Different Types of Databases. *My-Project-Management-Expert 17 July 2007* [Online] Available from: http://www.my-project-management-expert.com/different-types-of-databases.html [Accessed 06 February 2017]
15. IBM Corporation. History of IMS: Beginnings at NASA. *IBM website 24 October 2014* [Online] Available from: https://www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.imsintro.doc.intro/ip0ind0011003710.htm [Accessed 15 February 2017]
16. Abraham Silberschatz and Henry F Korth (1986). *Database System Concepts, New York, USA November 1986*. McGraw-Hill. ISBN 0-07-044752-7
17. Curt Monash. A bit of DB2 history, per IBM. *Software Memories.* 02 October 2008 [Blog Entry] Available from: http://www.softwarememories.com/2008/10/02/a-bit-of-db2-history-per-ibm/ [Accessed 11 February 2017]
18. Douglas K Barry. Relational Model Concepts. *Service Architecture Website, Barry & Associates, Inc.* [Online] Available from: http://www.service-architecture.com/articles/database/relational_model_concepts.html [Accessed 11 February 2017]
19. Chris Collins. History of SQL. *Chris Collins wordpress.* 20 May 2007 [Blog Entry] Available from: https://ccollins.wordpress.com/2007/05/20/history-of-sql/ [Accessed 14 February 2017]
20. Ling Lui and M Tamer Ozsu and Don Chamberlin (et al) (2009). *Encyclopedia of Database Systems, US, 2009*. Springer US. ISBN 978-0-387-39940-9 pp 2753-2760

21. Andrew Orlowski. Codd Almighty! How IBM cracked System R. *The Register website*. 20 November 2013. [Online] Available from: https://www.theregister.co.uk/2013/11/20/ibm_system_r_making_relational_really_real/?page=2 [Accessed 01 March 2017]
22. Dave Marshall. Remote Procedure Calls (RPC). *Programming in C UNIX System Calls and Subroutines using C website.* 05 January 1999. [Online] Available from: https://users.cs.cf.ac.uk/Dave.Marshall/C/node33.html [Accessed 30 January 2017]
23. James E. White. A High-Level Framework for Network-Based Resource Sharing. *Internet Request For Comments (RFC) 707, Stanford Research Institute, 23 December 1975* [Online] Available from https://tools.ietf.org/html/rfc707 [Accessed 04 February 2017]
24. Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *Association for Computing Machinery, Inc.* December 1983. [Online] Available from: http://www.textfiles.com/bitsavers/pdf/xerox/parc/techReports/CSL-83-7_Implementing_Remote_Procedure_Calls.pdf [Accessed 15 March 2017]
25. Lawrence M. Fisher. In Memoriam: Andrew Birrell 1951-2016. *Communications of the ACM website.* 09 December 2016. [Online] Available from: https://cacm.acm.org/news/210689-in-memoriam-andrew-birrell-1951-2016/fulltext [Accessed 15 March 2017]
26. Burnie Blakely, Harry Harris and Rhys Lewis (1995). *Messaging and Queueing Using the MQI, USA, 1995*. McGraw-Hill Inc. ISBN 0-07-005730-3
27. Nicholas Gall. The Origin (Coining) of the Term “Middleware”. *Nick Gall’s Weblog.* 02 November 2003 [Blog Entry] Available from: http://radio-weblogs.com/0126951/2003/11/02.html#a72 [Accessed 16 March 2017]
28. William A Ruh, Francis X Maginnis and William J Brown (2002). *Enterprise Application Integration: A Wiley Tech Brief, 2002*. John Wiley & Sons. ISBN 0471437867
29. Innovative Architects. Key Differences between ESB, EAI and SOA. *Innovative Architects website.* [Online] Available from: https://www.innovativearchitects.com/KnowledgeCenter/business-connectivity/ESB-EAI-SOA.aspx [Accessed 12 March 2017]
30. Jon Pierce. Integration Frameworks and Enterprise Integration Patterns. *Credera website.* 02 January 2014. [Blog Entry] Available from: https://www.credera.com/blog/technology-insights/java/integration-frameworks-enterprise-integration-patterns/ [Accessed 12 March 2017]
31. MuleSoft. What is application orchestration? *MuleSoft website.* 2017. [Online] Available from: https://www.mulesoft.com/resources/esb/what-application-orchestration [Accessed 01 March 2017]
32. Gregor Hohpe. *Enterprise Integration Patterns website*. [Online] Available from: http://www.enterpriseintegrationpatterns.com [Accessed 15 April 2017]
33. David Linthicum. *LinkedIn website*. [Online] Available from: https://www.linkedin.com/in/davidlinthicum [Accessed 10 March 2017]
34. Informit. David S Linthicum. *Informit website.* [Online] Available from: http://www.informit.com/authors/bio/19a6e9a3-73d7-4490-a15a-09b197abfb4e [Accessed 10 March 2017]
35. David S Linthicum (1999). *Enterprise Application Integration, USA, 2000*. Addison-Wesley Professional. ISBN 0-201-61583-5
36. Mike Clark, Peter Fletcher, Jeffrey J Hanson (et al) (2013). *Web Services Business Strategies and Architectures, 2013.* Apress. ISBN 1430253568 p39-50
37. SAS. ETL What it is and why it matters. *SAS website*. [Online] Available from: https://www.sas.com/en_us/insights/data-management/what-is-etl.html [Accessed 20 March 2017]
38. Open Database Alliance. The History of ODBC. *Open Database Alliance website.* [Online] Available from: http://www.opendatabasealliance.com/history.htm [Accessed 21 March 2017]

39. Patrick Grage (Javasoft). Introduction to JDBC. *Condor website*. [Online] Available from: http://condor.depaul.edu/elliott/513/projects-archive/DS513Fall98/project/Chess/JDBC.html [Accessed 21 March 2017]
40. Sun Microsystems. Sun ships JDK 1.1 – Javabeans included. *Sun Microsystems* [Online] Available from: https://web.archive.org/web/20080210044125/http://www.sun.com/smi/Press/sunflash/1997-02/sunflash.970219.0001.xml [Accessed 21 March 2017]
41. William Tse. Enterprise Application Integration. *Advanced Software Engineering, Unit Code: 3C05.* 2003. University College London. [Online] Available from: http://www0.cs.ucl.ac.uk/staff/ucacwxe/lectures/3C05-03-04/EAI.pdf [Accessed 09 March 2017]
42. Michael Richardson. Concept: Enterprise Application Integration. *Michael-richardson website, IBM Corporation*. 2006. [Online] Available from: http://www.michael-richardson.com/processes/rup_classic/modernize.legacy_evol/guidances/concepts/enterprise_application_integration_3CEC2399.html [Accessed 09 March 2017]
43. Ted Smalley Bowen. Neon updates messaging middleware. *Infoworld, Vol. 18, No. 30, 22 July 1996.* InfoWorld Media Group, Inc. ISSN 0199-6649. p8 [Online] Available from: https://books.google.co.uk/books?id=Ij0EAAAAMBAJ&pg=PA8 [Accessed 14 March 2017]
44. Matt. Happy 15th Birthday! A brief Technical History of IBM Integration Bus. *MQMATT, IBM DeveloperWorks, IBM Corportation* [Blog Entry] Available from: https://developer.ibm.com/integration/blog/2015/02/12/happy-15th-birthday-brief-technical-history-ibm-integration-bus/ [Accessed 27 March 2017]
45. Sybase. Introducing Financial Fusion tradeforce suite. *Computerworld, Vol. 36, No. 47, 18 Nov 2002*. IDG Enterprise. ISSN 0010-4841. p39 [Online] Available from: https://books.google.co.uk/books?id=uhXOKdElcJMC&pg=PA39 [Accessed 27 March 2017]
46. Grady Booch (1994). *Object-Oriented Analysis and Design with applications, California, USA, 1994*. The Benjamin/Cummings Publishing Company, Inc. ISBN 0-8053-5340-2
47. Andrew Severy. Teaching children computer skills to ensure they are never lost in forever loops and sub-routines. *Cambridge Independent*. 03 March 2017 [Online] Available from: http://www.cambridgeindependent.co.uk/education/schools/teaching-children-computer-skills-to-ensure-they-are-never-lost-in-forever-loops-and-sub-routines-1-4915731 [Accessed 6 April 2017]
48. Real Clear Politics. About RealClearPolitics. *RealClearPolitics website* [Online] Available from: http://www.realclearpolitics.com/about.html [Accessed 1 April 2017]
49. Stewart M. Clamen. About. *MRQE website.* [Online] Available from: http://www.realclearpolitics.com/about.html [Accessed 1 April 2017]
50. Yan Arrouye and Keith Mortensen. *Universal interface for retrieval of information in a computer system.* US 6847959 B1. 25 January 2000
51. Claudia Assis. Elon Musk tweets, Tesla shares leap. *Market Watch website.* 31 March 2015 [Online] Available from: http://www.marketwatch.com/story/elon-musk-tweets-tesla-shares-leap-2015-03-31 [Accessed 1 April 2017]
52. Jim Doree. RSS: A brief Introduction. *Journal of Manual & Manipulative Therapy, 2007*. [Online] Available from: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2565593/ [Accessed 11 April 2017]
53. Ajay Srivastava and Anant Bhargava. Understanding Application Servers. *TCS, Jan 2003*. [Online] Available from: http://hosteddocs.ittoolbox.com/AS030504.pdf [Accessed 3 April 2017]
54. IBM Corporation. WebSphere. *IBM Corporation website*. [Online] Available from: http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/websphere/ [Accessed 3 April 2017]
55. Tim Anderson. Introducing XML. *Tim Andersons IT writing* [Blog Entry] Available from: http://www.itwriting.com/xmlintro.php [Accessed 5 April 2017]
56. Chris Collins. A brief history of XML. *Chris Collins wordpress.* 3 March 2008 [Blog Entry] Available from: https://ccollins.wordpress.com/2008/03/03/a-brief-history-of-xml/ [Accessed 5 April 2017]

57. Eliotte Rusty Harold, W. Scott Means (2002). *XML in a Nutshell, June 2002*. O'Reilly Media, Inc. ISBN 978-0-596-00292-3
58. David Chappell (2004). *Enterprise Service Bus, June 2004*. O'Reilly Media, Inc. ISBN 0596006756
59. W Roy Schulte. Predicts 2003: SOA Is Changing Software. *Gartner Research website, ID G00111987, 09 December 2002*. [Online] Available from: https://www.gartner.com/doc/380364/predicts--soa-changing-software [Accessed 8 April 2017]
60. Jason Stampers. "ESB Inventor" Riddle Solved? *Computer Business Review website* [Blog Entry] Available from: http://www.cbronline.com/blogs/jason-stampers-blog/esb_inventor_ri/ [Accessed 8 April 2017]
61. IBM Corportation. IBM Completes Acquisition of Candle Corporation. *IBM Coportation*. 07 June 2004 [Online] Available from: http://www-03.ibm.com/press/us/en/pressrelease/7160.wss [Accessed 9 April 2017]
62. Andrew Binstock. Tibco Gets on the Enterprise Service Bus. *InfoWorld, Vol. 28, No. 50, 11 December 2006*. InfoWorld Media Group, Inc. ISSN 0199-6649 p44-45 [Online] Available from: https://books.google.co.uk/books?id=pzYEAAAAMBAJ&pg=PA45 [Accessed 9 April 2017]
63. Oxford Dictionary. service. *Oxford Dictionaries website*. [Online] Available from: https://en.oxforddictionaries.com/definition/service [Accessed 11 April 2017]
64. Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* University of California, 2000. [Online] Available from: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm [Accessed 11 April 2017]
65. Kin Lane. History of APIs. *API Evangelist*. 20 December 2012. [Blog Entry] Available from: http://apievangelist.com/2012/12/20/history-of-apis/ [Accessed 11 April 2017]
66. Alan Frye. A Bit of API History. *Design + Engineering*. 15 April 2015 [Blog Entry] Available from: https://www.benefitfocus.com/blogs/design-engineering/bit-api-history [Accessed 11 April 2017]
67. Readme. The History of REST APIs. *API Tips, readmeblog, Readme*. 15 November 2016. [Blog Entry] Available from: https://blog.readme.io/the-history-of-rest-apis/ [Accessed 11 April 2017]
68. Art Anthony. Tracking the growth of the API Economy. *Nordic APIS*. 18 April 2016 [Online] Available from: http://nordicapis.com/tracking-the-growth-of-the-api-economy/ [Accessed 11 April 2017]
69. Ethan Cerami (2002). *Web Services Essentials, February 2002*. O'Reilly Media, Inc. ISBN 0-596-00224-6
70. Martin Kalin. *Java Web Services: Up and Running, September 2013*. O'Reilly Media, Inc. ISBN 9781449373856
71. Dave Winer. XML-RPC for Newbies. *Dave Net*. 14 July 1998. [Blog Entry] Available from: http://scripting.com/davenet/1998/07/14/xmlRpcForNewbies.html [Accessed 12 April 2017]
72. Jeff Walsh. Microsoft spearheads protocol push. *InfoWorld Electric, 10 July 1998*. InfoWorld Media Group Inc. [Online] Available from: https://web.archive.org/web/19990914001234/http://www.infoworld.com/cgi-bin/displayStory.pl?980710.whsoap.htm [Accessed 12 April 2017]
73. Don Box. A Brief History of SOAP. *XML.com website*. 4 April 2001. [Online] Available from: http://www.xml.com/pub/a/ws/2001/04/04/soap.html [Accessed 12 April 2017]
74. Anne Thomas Manes (2003). *Web Services: A Manager's Guide, 2003*. Addison-Wesley Professional. ISBN 0321185773. p83-91
75. David Booth, Hugo Haas and Francis McCabe (et al). Web Services Architecture. *World Wide Web Consortium working group note, 11 February 2004*. [Online] Available from: https://www.w3.org/TR/ws-arch/#whatis [Accessed 9 April 2017]
76. IBM Corporation. *IBM FORTRAN Program Products for OS and the CMS Component of VM/370 General Information*, *New York, USA, July 1972*. IBM Corporation [Online] Available from: http://bitsavers.trailing-edge.com/pdf/ibm/370/fortran/GC28-6884-

0_IBM_FORTRAN_Program_Products_for_OS_and_CMS_General_Information_Jul72.pdf [Accessed 10 April 2017]
77. Oracle Corporation. FORTRAN 77 Language Reference. *Oracle Corporation*. [Online] Available from: http://docs.oracle.com/cd/E19957-01/805-4939/6j4m0vnc5/index.html [Accessed 19 March 2017]
78. Douglas Crockford. The JSON Saga. *YUI Library, 28 August 2011*. [Online] Available from: https://www.youtube.com/watch?v=-C-JoyNuQJs [Accessed 6 April 2017]
79. Progress. Working with JSON. *Progress Software Corporation*. [Online] Available from: https://documentation.progress.com/output/ua/OpenEdge_latest/index.html#page/dvjsn/benefits-of-json.html [Accessed 13 April 2017]
80. Roy T Fielding. *RE: [rest-discuss] RESTful representation of nouns?* Dr. Ernie Prabhakar. 5 April 2006 [Online] Available from: https://groups.yahoo.com/neo/groups/rest-discuss/conversations/topics/5841 [Accessed 13 April 2017]
81. John Mueller. Understanding SOAP and REST Basics and Differences. *Smartbear blog*. 8 January 2013. [Blog Entry] Available from: http://blog.smartbear.com/apis/understanding-soap-and-rest-basics/ [Accessed 13 April 2017]
82. Omed Habib. A quick primer on microservices. *Appdynamics blog*. 11 February 2016 [Blog Entry] Available from: https://blog.appdynamics.com/engineering/a-quick-primer-on-microservices/ [Accessed 15 April 2017]
83. Martin Fowler. Microservices a definition of this new architectural term. *Martin Fowler website*. 25 March 2014. [Online] Available from: https://martinfowler.com/articles/microservices.html [Accessed 15 April 2017]
84. Sam Newman. *Building Microservices: Designing Fine-Grained Systems, 2015*. O'Reilly Media, Inc. ISBN 1491950331
85. Jez Humble and David Farley (2010). *Continuous Delivery: Reliable Software Releases Through Build, Test and Deployment Automation, 27 July 2010*. Addison-Wesley. ISBN 0321670221
86. Martin Fowler. GOTO 2014 Microservices Martin Fowler. *GOTO Conferences*. 15 January 2015. [Online] Available from: https://www.youtube.com/watch?v=wgdBVIX9ifA&feature=youtu.be&t=13m10s [Accessed 10 February 2017]
87. W Roy Schulte and Yefim V Natis. "Service Oriented" Architectures, Part 1. *Gartner ID: G0029201*. 12 April 1996 [Online] Available from: https://www.gartner.com/doc/302868?ref=ddisp [Accessed 12 January 2017]
88. W Roy Schulte. "Service Oriented" Architectures, Part 2. *Gartner ID: G0029202*. 12 April 1996 [Online] Available from: https://www.gartner.com/doc/302869?ref=ddisp [Accessed 12 January 2017]
89. Yefim V Natis. "Service Oriented" Architectures Scenario *Gartner ID: G00114358*. 16 April 2003 [Online] Available from: https://www.gartner.com/doc/391595/serviceoriented-architecture-scenario [Accessed 12 January 2017]
90. Nicolai M Josuttis. *SOA in Pactice: The art of Distributed System Design, 2007*. O'Reilly Media, Inc. ISBN 059655155 p6-24
91. Heidi Buelow and Jayaram Kasi. *Getting Started with Oracle SOA Suite 11g R1, a Hands-on Tutorial: Fast Track Your SOA Adoption, Build a Service-oriented Composite Application in Just Hours!, October 2009*. Packt Publishing Ltd. ISBN 1847199798
92. David Linthicum. What Level is your SOA? *Sys-Con Media* [Blog Entry] Available from: http://davidlinthicum.sys-con.com/node/47277/mobile [Accessed 17 April 2017]
93. Microsoft. Chapter 1: Service Oriented Architecture (SOA). *Microsoft website* [Online] Available from: https://msdn.microsoft.com/en-us/library/bb833022.aspx [Accessed 17 April 2017]
94. Qusay H Mahmoud. Service-Oriented Architecture (SOA) and Web Services: The road to enterprise application integration (EAI). *Oracle Group*. April 2005. [Online] Available from: http://www.oracle.com/technetwork/articles/javase/soa-142870.html [Accessed 17 April 2017]

95. Douglas K Barry. Prior Service-Oriented Architecture Specifications. *Service Architecture Website, Barry & Associates, Inc.* [Online] Available from: http://www.service-architecture.com/articles/web-services/prior_service-oriented_architecture_specifications.html [Accessed 18 April 2017]
96. Kraig Brockschmidt. How OLE and COM Solve the problems of component software design. *Microsoft Corporation website*. May 1996. [Online] Available from: https://www.microsoft.com/msj/archive/S2EC.aspx [Accessed 18 April 2017]
97. Michi Henning. The Rise and Fall of CORBA. *Acmqueue, Volume 4, Issue 5*. 30 June 2006. [Online] Available from: http://queue.acm.org/detail.cfm?id=1142044 [Accessed 18 April 2017]
98. Object Management Group. History of CORBA. *Object Management Group*. [Online] Available from: http://www.omg.org/gettingstarted/history_of_corba.htm [Accessed 19 April 2017]
99. Judith Hurwitz, Robin Bloor, Marcia Kaufman and Dr Fern Halper (2009), *Service Oriented Architecture for Dummies, Indiana, USA, 2009*. Wiley Publishing Inc. ISBN 978-0-470-52549-4
100. Kim Clark. Integration architecture: Comparing web APIs with service-oriented architecture and enterprise application integration. *IBM DeveloperWorks*, *IBM Corporation.* 18 March 2015 [Blog Entry] Available from: https://www.ibm.com/developerworks/websphere/library/techarticles/1503_clark/1305_clark.html [Accessed 19 April 2017]
101. The Open Group. The Open Group Service Integration Maturity Model (OSIMM) Version 2 – The Model. The Open Group Website, 2016. [Online] Available from: http://www.opengroup.org/soa/source-book/osimmv2/p2.htm [Accessed 20 April 2017].
102. Kunal Mittal. Build you SOA, Part 2: The Service-Oriented Architecture Maturity Model. *IBM DeveloperWorks*, *IBM Corporation.* 18 March 2015 [Blog Entry] Available from: https://www.ibm.com/developerworks/library/ws-soa-method2/ [Accessed 19 April 2017]
103. Christina Warren. Twitter's API Update Cuts Off Oxygen to Third-Party Clients. *Mashable UK.* 16 August 2012. [Online] Available from: http://mashable.com/2012/08/16/twitter-api-big-changes/#lJ.JmI5QCuqF [Accessed 20 April 2017]
104. Charles Arthur. Google Keep? It'll probably be with us until March 2017 – on average. *The Guardian*. 22 March 2013 [Online] Available from: https://www.theguardian.com/technology/2013/mar/22/google-keep-services-closed [Accessed 20 April 2017]
105. Jemima Kiss. Google Reader RSS feed aggregator to be retired. *The Guardian*. 14 March 2013. [Online] Available from: https://www.theguardian.com/technology/2013/mar/14/google-reader-rss-retired [Accessed 20 April 2017]
106. Codd, E. (1970). *A Relational Model of Data for Large Shared Data Banks*. Ph.D. IBM Research Laboratory.
107. Haigh, T. (n.d.). *Charles W. Bachman - A.M. Turing Award Winner*. [online] Amturing.acm.org. Available at: https://amturing.acm.org/award_winners/bachman_1896680.cfm [Accessed 1 Apr. 2019].
108. Tools.ietf.org. (1976). *RFC 707 - High-level framework for network-based resource sharing*. [online] Available at: https://tools.ietf.org/html/rfc707 [Accessed 1 Apr. 2019].
109. NASA. (2004). *Excerpt: 'Special Message to the Congress on Urgent National Needs'*. [online] Available at: https://www.nasa.gov/vision/space/features/jfk_speech_text.html [Accessed 1 Apr. 2019].
110. Broadbent, V. (2005). How IBM misjudged the PC revolution. *BBC News*. [online] Available at: http://news.bbc.co.uk/1/hi/business/4336253.stm [Accessed 1 Apr. 2019].
111. Green, T. (2017). *IBM's New Mainframe Is a Security Powerhouse -- The Motley Fool*. [online] The Motley Fool. Available at: https://www.fool.com/investing/2017/07/17/ibms-new-mainframe-is-a-security-powerhouse.aspx [Accessed 1 Apr. 2019].
112. Kamp, P. (2016). *Jails – High value but....* [online] Phk.freebsd.dk. Available at: http://phk.freebsd.dk/sagas/jails/ [Accessed 1 Apr. 2019].
113. Kamp, P. and Watson, R. (n.d.). *Jails: Confining the omnipotent root.*. [online] Phk.freebsd.dk. Available at: http://phk.freebsd.dk/pubs/sane2000-jail.pdf [Accessed 1 Apr. 2019].

114. Bagnato, A., Indrusiak, L., Quadri, I. and Rossi, M. (2014). *Handbook of research on embedded system design*. IGI Global., pp.379 - 386.
115. GitHub. (2019). *google/lmctfy*. [online] Available at: https://github.com/google/lmctfy/blob/master/README.md [Accessed 1 Apr. 2019].
116. Vaughan-Nichols, S. (2019). *Docker 1.0 brings container technology to the enterprise | ZDNet*. [online] ZDNet. Available at: https://www.zdnet.com/article/docker-1-0-brings-container-technology-to-the-enterprise/ [Accessed 1 Apr. 2019].
117. Clark, J. (2014). *Google: 'EVERYTHING at Google runs in a container'*. [online] Theregister.co.uk. Available at: https://www.theregister.co.uk/2014/05/23/google_containerization_two_billion/ [Accessed 1 Apr. 2019].
118. Pearson, D., Gibson, G. and Katz, R. (2019). *A Case for Redundant Arrays of Inexpensive Disks (RAID)*. University of California.
119. Portier, B. (2014). *Always On: Business Considerations for Continuous Availability*. [ebook] IBM, p.ww.redbooks.ibm.com. Available at: http://www.redbooks.ibm.com/redpapers/pdfs/redp5090.pdf [Accessed 1 Apr. 2019].
120. Brewer, E. (2012). *CAP Twelve Years Later: How the "Rules" Have Changed*. [online] InfoQ. Available at: https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed [Accessed 1 Apr. 2019].
121. Abadi, D. (2010). *Problems with CAP, and Yahoo's little known NoSQL system*. [online] Dbmsmusings.blogspot.com. Available at: http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html [Accessed 1 Apr. 2019].
122. Abadi, D. (2012). *Consistency Tradeoffs in Modern Distributed Database System Design*. [ebook] Yale University. Available at: http://cs-www.cs.yale.edu/homes/dna/papers/abadi-pacelc.pdf [Accessed 1 Apr. 2019].
123. Dewan, P. (2006). *Synchronous vs Asynchronous*.
124. Gardner, D. (1999). *Progress delivers messaging middleware based on JMS*. [ebook] InfoWorld, p.24. Available at: https://books.google.co.uk/books?id=rk4EAAAAMBAJ&pg=PA24&dq=sonicMQ&hl=en&sa=X&ved=0ahUKEwias7GVz83aAhXkLsAKHclKDDAQ6AEIKTAA#v=onepage&q=sonicMQ&f=false [Accessed 1 Apr. 2019].
125. ZDNet. (2000). *Bridging Interoperability | ZDNet*. [online] Available at: https://www.zdnet.com/article/bridging-interoperability/ [Accessed 1 Apr. 2019].
126. Kramer, J. (2009). *Advanced Message Queuing Protocol (AMQP) | Linux Journal*. [online] Linuxjournal.com. Available at: https://www.linuxjournal.com/article/10379 [Accessed 1 Apr. 2019].
127. Docs.oasis-open.org. (2012). *OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0, Part 2: Transport*. [online] Available at: http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-transport-v1.0-os.html#section-performatives [Accessed 1 Apr. 2019].
128. Docs.oasis-open.org. (2012). *OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0, Part 3: Messaging*. [online] Available at: http://docs.oasis-open.org/amqp/core/v1.0/amqp-core-messaging-v1.0.html#type-source [Accessed 1 Apr. 2019].
129. Apache Kafka. (n.d.). *Apache Kafka*. [online] Available at: https://kafka.apache.org/intro [Accessed 1 Apr. 2019].
130. Gutierrez, D. (2016). *A Brief History of Kafka, LinkedIn's Messaging Platform - insideBIGDATA*. [online] insideBIGDATA. Available at: https://insidebigdata.com/2016/04/28/a-brief-history-of-kafka-linkedins-messaging-platform/ [Accessed 1 Apr. 2019].
131. Honeywell (n.d.). *What the heck is Electronic Mail?*. [image] Available at: https://multicians.org/thvv/vvimg/honeywell_email.jpg [Accessed 1 Apr. 2019].
132. Van Vleck, T. (2001). *The History of Electronic Mail*. [online] Multicians.org. Available at: https://multicians.org/thvv/mail-history.html [Accessed 1 Apr. 2019].
133. Van Vleck, T. (2008). *Documentation and Source for Early Electronic Mail and Messaging*. [online] Multicians.org. Available at: https://multicians.org/thvv/mail-details.html [Accessed 1 Apr. 2019].
134. Multicians.org. (n.d.). *Multics Networking and Communications*. [online] Available at: https://multicians.org/mx-net.html [Accessed 1 Apr. 2019].
135. Padlipsky, M. (2000). *And They Argued All Night....* [online] Multicians.org. Available at: https://multicians.org/allnight.html [Accessed 1 Apr. 2019].
136. Tomlinson, R. (n.d.). *The First Email*. [online] Openmap.bbn.com. Available at: http://openmap.bbn.com/~tomlinso/ray/firstemailframe.html [Accessed 1 Apr. 2019].

137. Ibm.com. (n.d.). *IBM Knowledge Center*. [online] Available at: https://www.ibm.com/support/knowledgecenter/en/SSQPD3_2.6.0/com.ibm.wllm.doc/ov_assuredmessaging.html [Accessed 1 Apr. 2019].
138. European Commission. (n.d.). *EU Data protection rules* [online] ec.europa.eu. Available at: https://ec.europa.eu/commission/priorities/justice-and-fundamental-rights/data-protection/2018-reform-eu-data-protection-rules/eu-data-protection-rules_en [Accessed 1 Sept. 2019].
139. n/a
140. Lowy, J. (2009). *Working with the .NET Service Bus*. [online] Msdn.microsoft.com. Available at: https://msdn.microsoft.com/en-us/magazine/dd569756.aspx [Accessed 1 Apr. 2019].
141. Jennings, R. (2011). *Cloud Computing with the Windows Azure Platform*. New York, NY: John Wiley & Sons, p.273.
142. Businesswire.com. (2011). *FuseSource Fuse ESB Cited as a Leader in Open Source ESBs by Independent Research Firm*. [online] Available at: https://www.businesswire.com/news/home/20110428006585/en/FuseSource-Fuse-ESB-Cited-Leader-Open-Source [Accessed 1 Apr. 2019].
143. Khurana, A. (n.d.). *How Portlets are different from traditional Servlets?*. [online] Webportalclub.com. Available at: http://www.webportalclub.com/2011/02/servlets-vs-portlets.html [Accessed 1 Apr. 2019].
144. IBM (2001). *IBM Merges WebSphere Portal Server And Lotus K-station Into Single Portal, Announces New Portal Partners*. [online] Available at: https://www-03.ibm.com/press/us/en/pressrelease/1132.wss [Accessed 1 Apr. 2019].
145. Scannell, E. (2001). *IBM rolls out WebSphere Portal Server*. [online] Computerworld. Available at: https://www.computerworld.com/article/2584568/ibm-rolls-out-websphere-portal-server.html [Accessed 1 Apr. 2019].
146. Oracle Fusion Middleware FAQ. (2007). [ebook] Oracle WebCenter. Available at: https://www.oracle.com/technetwork/middleware/webcenter/owcs-10132-external-faq-128025.pdf [Accessed 1 Apr. 2019].
147. Clark, K. and Petrini, B. (2012). *Reference guide to integration characteristics*. [online] Ibm.com. Available at: https://www.ibm.com/developerworks/websphere/techjournal/1201_clark/1201_clark.html [Accessed 1 Apr. 2019].
148. Eugster, P., GUERRAOUI, R., Felber, P. and Kermarrec, A. (2003). *The Many Faces of Publish/Subscribe*.
149. Emerson, B. (2016). *'Atlanta in 50 Objects' at Atlanta History Center*. [online] The Atlanta Journal-Constitution. Available at: https://www.myajc.com/entertainment/arts--theater/atlanta-objects-atlanta-history-center/IBMM4VbEDHax8gxxo1SD6J/ [Accessed 1 Apr. 2019].
150. Viaene, E. (2015). *The 5 Business Benefits of the Twelve Factor App*. [online] Linkedin.com. Available at: https://www.linkedin.com/pulse/5-business-benefits-twelve-factor-app-edward-viaene/ [Accessed 1 Apr. 2019].
151. Gray, J. (1981). *The Transaction Concept: Virtues and Limitations*. [ebook] Cupertino: Tandem Computers Incorporated. Available at: http://jimgray.azurewebsites.net/papers/thetransactionconcept.pdf [Accessed 1 Apr. 2019].
152. McJones, P. (1998). *Jim Gray - A.M. Turing Award Laureate*. [online] Amturing.acm.org. Available at: https://amturing.acm.org/award_winners/gray_3649936.cfm [Accessed 1 Apr. 2019].
153. Gray, J. and Reuter, A. (1992). *Transaction Processing: Concepts and Techniques*. Elsevier.
154. G. E. Kaiser, D. E. Perry and W. M. Schell, "Infuse: fusing integration test management with change management," [1989] Proceedings of the Thirteenth Annual International Computer Software & Applications Conference, Orlando, FL, USA, 1989, pp. 552-558. doi: 10.1109/CMPSAC.1989.65147
155. Bias, R. (2016). *The History of Pets vs Cattle and How to Use the Analogy Properly*. [online] Cloudscaling. Available at: http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle/ [Accessed 1 Apr. 2019].
156. Clark, K. (2017). *"Cattle not pets" for IBM Integration Bus - new article - IBM Integration*. [online] Developer.ibm.com. Available at: https://developer.ibm.com/integration/blog/2017/09/14/cattle-not-pets-for-ibm-integration-bus-new-article/ [Accessed 1 Apr. 2019].
157. Bornert, C. and Clark, K. (2017). *Cattle not pets: Achieving lightweight integration with IBM Integration Bus*. [online] developerworks. Available at: https://www.ibm.com/developerworks/library/mw-1708-bornert/index.html [Accessed 1 Apr. 2019].

158. Clark, K. (2018). *Moving to a lightweight, agile integration architecture*. [online] DeveloperWorks. Available at: https://developer.ibm.com/articles/cl-lightweight-integration-2/ [Accessed 1 Apr. 2019].
159. Papp, A. (2018). *The History of Kubernetes on a Timeline*. [online] RisingStack Engineering. Available at: https://blog.risingstack.com/the-history-of-kubernetes/ [Accessed 1 Apr. 2019].
160. Borenstein, N. and Freed, N. (1992). *MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies*. [online] Tools.ietf.org. Available at: https://tools.ietf.org/html/rfc1341 [Accessed 1 Apr. 2019].
161. Global Music Report 2018. (2018). [ebook] ifpi. Available at: https://www.ifpi.org/downloads/GMR2018.pdf [Accessed 1 Apr. 2019].
162. Horowitz, B. (2010). *Why We Prefer Founding CEOs*. [online] Andreessen Horowitz. Available at: https://a16z.com/2010/04/28/why-we-prefer-founding-ceos/ [Accessed 1 Apr. 2019].
163. Bryan, A. (2014). *DIGITAL PIRACY: NEUTRALISING PIRACY ON THE DIGITAL WAVES*. University of Plymouth.
164. Essays, UK. (November 2013). The History Of The Act Of Torrenting Information Technology Essay. Retrieved from https://www.uniassignment.com/essay-samples/information-technology/the-history-of-the-act-of-torrenting-information-technology-essay.php [Accessed 1 Apr. 2019].
165. Bailey, J. (2017). *The Long, Slow Decline of BitTorrent - Plagiarism Today*. [online] Plagiarism Today. Available at: https://www.plagiarismtoday.com/2017/06/01/the-long-slow-decline-of-bittorrent/ [Accessed 1 Apr. 2019].
166. Lee, D. (2014). *Pirate Bay goes offline after raid*. [online] BBC News. Available at: https://www.bbc.co.uk/news/technology-30411782 [Accessed 1 Apr. 2019].
167. BBC. (2016). *BBC Four - Storyville, The Pirate Bay, The Pirate Bay founders stand trial*. [online] Available at: https://www.bbc.co.uk/programmes/p04b5dmp [Accessed 1 Apr. 2019].
168. Brown, A. (2019). *Pirate Bay WARNING: Thousands SUED for downloading for FREE, as legal action intensifies*. [online] Express.co.uk. Available at: https://www.express.co.uk/life-style/science-technology/917847/The-Pirate-Bay-Torrent-Warning-Download-UK [Accessed 1 Apr. 2019].
169. Heisler, G. (2000). What's next for napster. *TIME*, [online] (Vol. 156), p.Front Cover. Available at: http://content.time.com/time/covers/0,16641,20001002,00.html [Accessed 1 Apr. 2019].
170. Engadget. (2016). *Public Access - The History of Youtube*. [online] Available at: https://www.engadget.com/2016/11/10/the-history-of-youtube/ [Accessed 1 Apr. 2019].
171. Advanced video coding for generic audiovisual services. (2003). [ebook] ITU-T. Available at: http://handle.itu.int/11.1002/1000/6312 [Accessed 1 Apr. 2019].
172. ISO, (2009). *Information technology — Coding of audio-visual objects — Part 3: Audio PREVIEW*. [online] Available at: https://webstore.iec.ch/preview/info_isoiec14496-3%7Bed4.0%7Den.pdf [Accessed 1 Apr. 2019].
173. Ho, A. and Li, S. (2016). *Handbook of Digital Forensics of Multimedia Data and Devices, Enhanced E-Book*. John Wiley & Sons.
174. Postel, J. (1980). *RFC 768 - User Datagram Protocol*. [online] Tools.ietf.org. Available at: https://tools.ietf.org/html/rfc768 [Accessed 1 Apr. 2019].
175. Schulzrinne, H., Casner, S., Frederick, R. and Jacobson, V. (2003). *RFC 3550 - RTP: A Transport Protocol for Real-Time Applications*. [online] Tools.ietf.org. Available at: https://tools.ietf.org/html/rfc3550 [Accessed 1 Apr. 2019].
176. McGrath, G. (2013). *Basics of Streaming Protocols*. [online] Garymcgath.com. Available at: http://www.garymcgath.com/streamingprotocols.html [Accessed 1 Apr. 2019].
177. Dwyer, E. (2017). *2017 on Netflix - A Year in Bingeing*. [online] Netflix Media Center. Available at: https://media.netflix.com/en/press-releases/2017-on-netflix-a-year-in-bingeing [Accessed 1 Apr. 2019].
178. Telescope A look at the nations viewing habits. (2017). [ebook] TV Licensing. Available at: https://www.tvlicensing.co.uk/ss/Satellite?blobcol=urldata&blobheadername1=content-type&blobheadervalue1=application%2Fpdf&blobkey=id&blobtable=MungoBlobs&blobwhere=1370006458226&ssbinary=true [Accessed 1 Apr. 2019].
179. n/a
180. Telegraph (2017). The story behind the world's first cashpoint. [online] Available at: https://www.telegraph.co.uk/personal-banking/current-accounts/story-behind-worlds-first-cashpoint/ [Accessed 20 Feb. 2019].
181. BBC (2007). The man who invented the cash machine. [online] Available at: http://news.bbc.co.uk/1/hi/business/6230194.stm [Accessed 20 Feb. 2019].

182. Sherry, N. (2018). *3 ways IBM MQ helps with daily life*. [online] Cloud computing news. Available at: https://www.ibm.com/blogs/cloud-computing/2018/07/26/3-ways-ibm-mq-helps-daily-life/ [Accessed 1 Apr. 2019].
183. Google Developers. (n.d.). *Developers Guide*. [online] Available at: https://developers.google.com/protocol-buffers/docs/overview [Accessed 1 Apr. 2019].
184. Google Developers. (n.d.). *Frequently Asked Questions*. [online] Available at: https://developers.google.com/protocol-buffers/docs/faq [Accessed 1 Apr. 2019].
185. IBM (2002). *IBM Completes Acquisition of CrossWorlds Software*. [online] Available at: https://www-03.ibm.com/press/us/en/pressrelease/947.wss [Accessed 1 Apr. 2019]