Service Virtualisation

Eamonn Lawler

The pioneering aeronautic accomplishments of Orville and Wilbur Wright are well documented in our history books, but as with many histories, we know less about the hard graft that preceded their success. To begin with, trial and error was the only means of getting to their desired outcome – a flying machine. This proved to be an expensive and inefficient means of development; each trial depending on a complete flying machine, even if they were interested in testing only one of its parts and each error resulting in costly setbacks. Their approach was a form of waterfall methodology – lengthy sequences of design, development and testing. When testing proved the need for further design and development, they had to go right back to the beginning and start the whole process again. Agile, it was not.

This inefficiency motivated the Wright brothers to come up with a better method; one that would enable them to develop and test components without depending on an entire flying machine. Their solution was an engine-powered wind tunnel. They could then synthetically recreate the behaviour of something on which their system depended – airspeed. Being able to do this on demand meant they could reduce the cost and effort associated with each development-test iteration. Whereas previously, they could only subject a new wing design to airspeed once they had a complete airplane to fly, the wind tunnel allowed them to test the behaviour of wing prototypes in isolation.

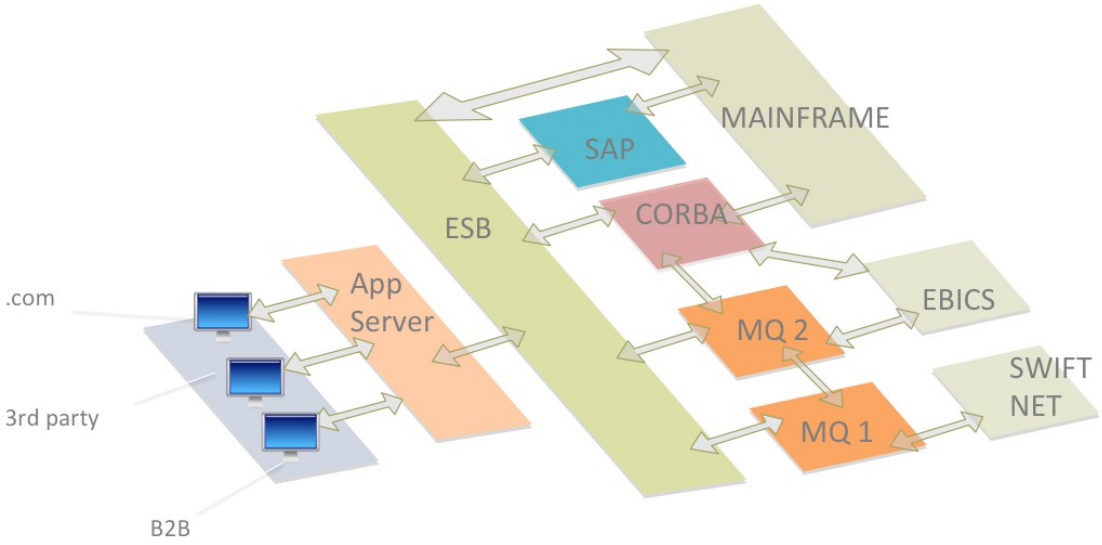
Many software development practices could learn from this `testing in isolation' paradigm; quite often we rely on complete end-to-end environments, even to test individual components. For example, most organizations in the financial sector maintain huge non-production environments for development and testing purposes. Their footprints are typically 3 to 4 times the size of their actual production environment.

The cost and effort associated with provisioning and maintaining these non-production environments is a problem in itself. Worse still, delays and outages related to environment and test data result in lengthened project timescales and restrict the amount of testing that may be done pre-production. The dependency on non-production infrastructure and test data is analogous to the Wright Brother's dependency on a complete airplane.

This paper is a look at the emerging pattern of Service Virtualisation, and how it has been used over the last few years by software houses to mitigate the need for non-production environments. The adoption of Service Virtualisation brings about a step change in efficiency and quality of Software Development Life Cycles (SDLCs).

Challenges of Developing and Testing in Composite Systems

Today, most business applications are some form of composite application, i.e. some pattern of distributed, heterogeneous and autonomous participating components. The diagram is an abstract example of such a system.



A business process that runs on this system will depend on the whole composite system. Think of an online transaction in a retail .com infrastructure, or payment processing in a bank infrastructure; in order for the business process to successfully complete, all participating components within the infrastructure must be present and correct. If a single component is not functioning, the entire process fails.

This feature makes software development and testing of composite systems very difficult. Constraints within such systems include the following:

Dependency on external services

A composite application may depend on access to a third party Web service or to transactions running on a mainframe. Access to this third party service is required for development purposes (test driven development or unit testing) and testing purposes (functional, regression, performance or load testing). Non-production instances of third party services, that are fit for purpose, are rare. Non-production services tend to suffer from:

- *Availability* – developers and testers may have scheduled access to a mainframe, for example, which means it is available to them only during

scheduled times. Testers may also suffer from unscheduled downtime of systems, which impacts how much testing they can perform.

- *Unrepresentative of production* – for example, a common shortcoming of third party non-production systems is that they don't realistically mirror production response times.
- *Test coverage* – they will typically support only a narrow band of happy path test cases, and perhaps a few boundary conditions.

Given these inherent limitations, the unfortunate reality is that much testing gets done in production. Production is often the first time a system is exposed to behaviour such as increased latency when the system is under high load, or edge conditions, which are difficult to reproduce in non-production. We must accept that, with limited non-production test coverage, we will discover issues in production.

Cost and time to provide environments

Setting up a replica of a production system is not easy. Someone must provision hardware and software licences, and maintain them. If mainframes are involved, add MIPS usage to that cost. I have worked with a number of organisations where the cost of non-production environments ran into tens of millions of pounds.

Environment availability

Environment availability issues are a plague for testing teams. This is especially prevalent where composite applications are being developed, for which a single user story involves interaction with many systems; it takes only one system to be down to break the testing scenario. I have worked in dedicated test centres where environment availability was less than 50%. That means teams of testers are idle at least half the time!

Services not yet developed

Think of a Service-Oriented Architecture in which some of the components are themselves under development. How do we test an end-to-end scenario when parts of the system are not yet in place? Traditionalists may suggest that this is a moot point, because end-to-end testing only begins once we have a complete system. But given that the functional behaviour in a SOA is exhibited by the system as a whole, we may wish to test this functional behaviour even before some of the components are available.

Test data

Test coverage is dependent upon test data. In general, for every test scenario, there must be test data to support that scenario. Ideally, we would wish for 100% test coverage. In reality, most organisations achieve about 10% of this, partly due to the fact that test data is hard to create.

Dr. Stephan Murer, Visiting Professor of Software Engineering at Oxford University, described[1] the challenges of testing in `very large information systems' (based on his experience as Chief Architect at a large bank). He made a point of challenging the commonly held belief that production data makes good test data (testing cycles commonly involve a collection of production data, which is then used to stage a test). The premise is that, if you include enough production data, you get good test coverage. Dr. Murer cited a catastrophic failure in a core mainframe module, which led to billions of dollars being temporarily `lost'. The cause was incorrect handling of 29th February in a leap year. This mainframe module had recently undergone a revision and although the testing cycle had included a rigorous test using three months' worth of production data, it did not include the leap year edge case. His point – production data is bad test data; you may get reasonably good test coverage, but you should not assume it's 100%.

In composite systems, the distributed nature of components means that test data must often be synchronised across multiple components. For example, a process running on an enterprise service bus will interact with several systems. If that process has a customer context, then each of the participating systems must also know about that customer. So, test data management is more complex due to a need to synchronise test data across multiple systems. Nicolai Josuttis (author of `SOA in Practice') makes this point, *`For example, providing distributed test data and distributed debugging can be really challenging.'*

Contention issues compound these headaches; we spend considerable effort to provision test data in our non-production environment. One tester executing a user story may then render some of this test data unusable for another tester (since the business process may alter the state of the test data).

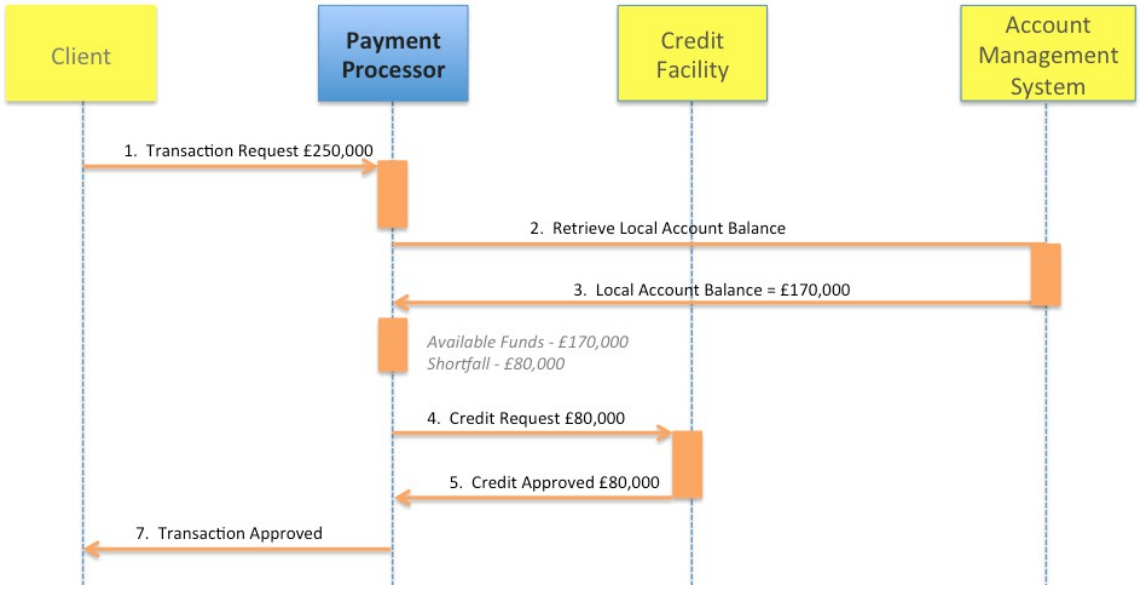
In summary, the evolution of software systems has taken us to the point where composite systems are the most common form of computer system under development. And we have touched on some of the unique constraints, specific to composite systems, that hinder software development and testing.

The software industry has yet to address these constraints properly. Many software houses are at the stage the Wright brothers were before their wind tunnel.

How can Service Virtualisation help?

Service Virtualisation is the practice of realistically simulating the behaviour of systems in a similar manner to mocking or stubbing techniques. It is `*service*' virtualisation, because we create an *interface*, rather than an actual system. The virtual service completely abstracts the details of the real system, which it is virtualising.

The first question in a Service Virtualisation engagement should be `what is the system *in scope*'. Which part(s) are we are interested in developing and/or testing, and by inference, what are the downstream systems, which we depend upon, but are not responsible for. Consider a real-life example below. This illustrates a payments processing user story from a large bank. The process depends upon the participation of four entities.



The team I worked with was responsible for development and testing of the payment processing component (highlighted in blue). Three other teams, located in different geographic regions, managed the other systems. In order to perform testing of a user story, the payment processing team depends upon the participation of the three other teams. This dependency was proving to be a prohibitive constraint (given the time zone differences and the complexities of synchronising customer payment test data across all four systems). The `in scope' system here is the payments processing system. The other three systems represent dependencies.

So, the objective was to create virtual services, which would take the place of these three systems, and thereby enable unconstrained execution of user stories through the in-scope system, the payment processor.

Why is Service Virtualisation Better than Stubbing?

At this point, it may sound like we're describing stubbing or mocking techniques; and indeed, the end goal is the same – overcoming environmental constraints by creating *simulated systems*, which can be used in the place of real systems.

`Stubbing' or `mocking' typically refers to a practice of developers creating stubs by coding, or testers using tools such as SoapUI. The term `stub' has a connotation of

something trivial and lightweight, but in reality, if a stub is to be useful for functional testing, it should support a broad range of test cases. If it is to be useful for non-functional testing, it should be scalable to support the necessary volumes whilst keeping to response time Service Level Agreements (SLAs). Creating such a stub is not a trivial matter, and this results in a common problem with this approach – developers spend considerable amounts of time creating and maintaining stubs, rather than developing product.

In some circumstances where basic Web service functionality is required, SoapUI (and other such software) can be a developer's best friend, enabling easy provisioning of mocks from service descriptions.

Composite systems are typically heterogeneous, with many different technologies in play. Think of an ESB with a variety of end-points; EDI, JMS, MQ, or various flavours of Web services (SOAP, HTTP etc). Manually creating stubs for all such end-points is impractical, and tools such as SoapUI have limited capabilities outside standard Web services. Secondly, composite systems are distributed systems with many end-points; having stubs or mocks work in a synchronised manner (i.e. to support shared state etc.) is impractical when you consider the challenge of synchronising test data across multiple end-points.

A case in point; one of the world's largest financial institutions runs a webMethods integration hub, which talks to over 70 back-end systems over MQ and Web services (soap/https). The goals for this SOA architecture are the usual `standards' (integrating disparate systems, business agility to offer new services to market etc.). They employed the services of a large outsourcing company to create stubs, which would be used in place of the real back-end systems for development and testing purposes. After two years, and having spent a fortune on several developer teams manually building these stubs, they had little success. The stubs were labour intensive to create, were high maintenance, and proved to have low re-usability. The Project Lead for this company's Testing Centre of Excellence invited a Service Virtualisation vendor to conduct a proof of concept. The Project Lead claimed that, within two weeks, the proof of concept team had surpassed the efforts of the developers' two years' worth of effort. This is not to say the developers were in some way incompetent, but serves to illustrate the efficiency and quality gains realised by using a purpose built Service Virtualisation tool set.

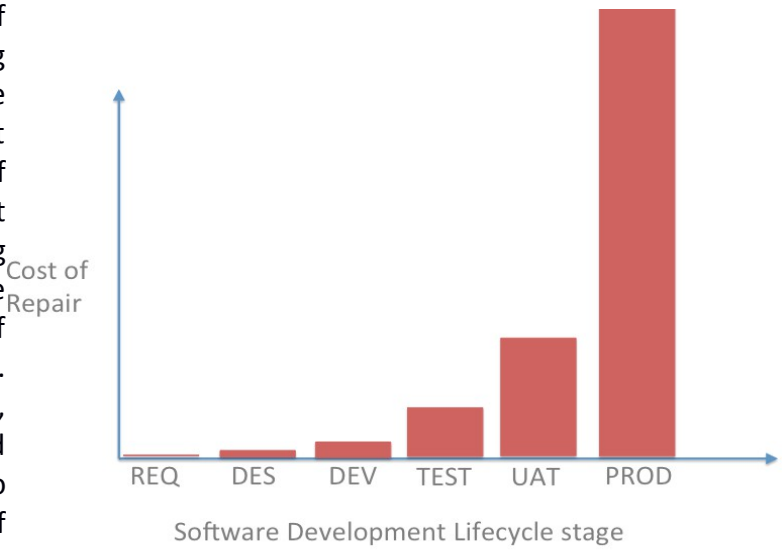
This bank now has a small handful of people in a Testing Centre of Excellence, responsible for providing virtual services for the entire global operations of the bank. The fact that a handful of employees can provide follow-the-sun support to meet the needs of development and testing teams throughout the globe, illustrates that this is a far more efficient method than manual stubbing.

Very few software houses would choose to build their own data management system, rather than purchase a commoditised DBMS, for obvious reasons. Similarly, we'll see over the next few years that very few will be building and maintaining stubs.

Benefits of Service Virtualisation Throughout the SDLC

A typical SDLC incorporates stages we are familiar with – design, development, functional testing, system integration testing, user acceptance testing and production. In a perfect world, we could progress directly from development to production. In the real world, the intermediary stages are necessary to identify and remove unwanted features from our system before it gets deployed to production.

Most readers will have seen some variation of this chart[2], which plots the cost of addressing software issues, relative to where they are discovered in a SDLC. For example, if a defect is found in development, the cost of addressing the defect is low (since the impact is localised to development). Discovering issues in the testing phase is different; now we are warranting the time of a separate group of individuals, whose job it is to test. Furthermore, each time they discover a defect, there is some process of logging and communicating steps-to-reproduce back to development, and further iterative cycles of development, and regression testing etc. Clearly it is *much* more expensive to discover defects in testing than in development. Similarly, the cost increases in some exponential manner for each stage we progress through. Discovering defects in production is worst case scenario and the impact of this is context dependent (loss of business for online retailers, reputation damage for online banks, loss of life for safety critical systems etc.). This graph often prompts debate amongst cynics and skeptics about whether there really is an exponential trend. However, there is no argument that such an upwards trend exists in some form; it is *far* more cost-effective to address software defects to the left of this graph than it is to do so on the right.



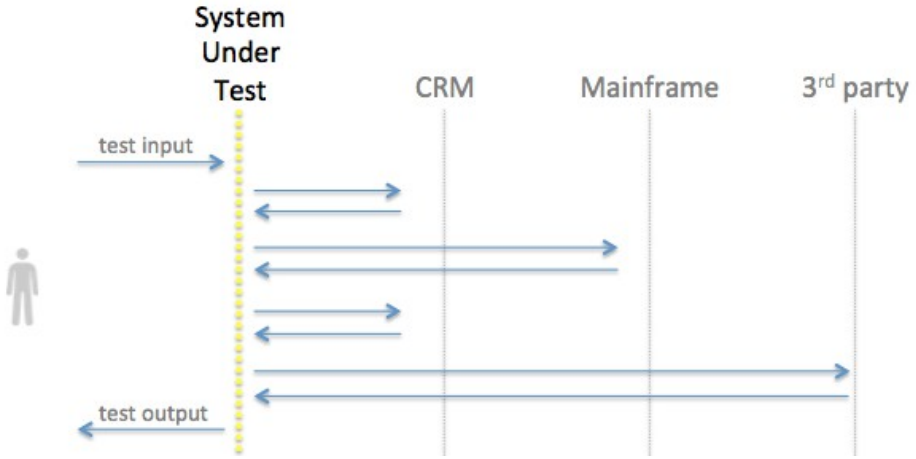
And this is what Service Virtualisation is all about – enabling us to detect software issues earlier. Practitioners in the Service Virtualisation field speak of a `*shift left'* pattern – detecting issues *earlier* in the development cycle.

It is useful to understand specific use cases for virtual services throughout the SDLC:

- **Development**
  - **Unit testing** – developers may test their component against downstream systems, which would otherwise be unavailable to them. The virtual services may be hosted on a shared server for all employees in an organisation, for a single development team, or even on a developer's own workstation. Given that the footprint of virtual services is minimal, it is common for a developer to have a suite of virtual services running in a virtual service environment locally on his/her workstation/laptop.
  - **Test Driven Development** – virtual downstream systems support all required functional (happy path, negative, boundary, edge case etc.) and non-functional (delayed response, no response etc.) testing. This makes it possible to have a test rig available to support development against specification.
  - **Prototypes for Development** – virtual services are, in effect, malleable models, which may be used in a non-production environment. By playing with these models, the real implementation may be derived from their use. The CIO of Best Buy (world's largest multi-channel consumer electronics retailer) speaks[3] of how they used virtual services in this manner. By setting up models of services, they could analyse how those virtual services were *actually* consumed. In playing with these prototypes, they learned that some services were being called too many times, some not enough etc., and this helped them iteratively define the granularity of services. Such a model-driven approach was only possible with the use of virtual services.
  - **Offshore Development** – There is much scope for debate on where the balance between benefits and costs lies when it comes to offshore development. Undoubtedly, one disadvantage is that an offshore team has limited, or no, access to its client's infrastructure. Virtual services are commonly used to provide offshore development teams with complete virtual environments to play with. The measurable benefit here is in terms of code coming back from offshore teams with fewer defects, since they have had opportunity to test against virtual environments.
  - **Agile** – It seems that everyone loves the idea of agile. In many organisations, that's all it is – an idea. Agile endorses patterns such as short iterative development cycles, the use of working prototypes, minimal formal design structure, Joint Application Development workshops etc. This is all achievable for standalone monolithic applications. But how do you do agile in composite systems? When one development team is dependent on a code-drop from another, that's a blocker to agile. When separate teams work in isolation and then come together for system integration, the results are not always pretty. David Norton, a Gartner Research Director, spoke[4] of inhibitors to agile, principally interfaces and data. He sternly advised the use of virtual

services tools – "*…as an agile guy, I don't like talking about tools, but this is really worth its weight in gold*".

- **Functional Testing**
  - **Negative/boundary testing** – Using virtual services means we are free to support a range of test scenarios that would not be possible with a physical system. Orville and Wilbur presumably learned about a wing's lift coefficient, and how increasing the tilt of the wing to a point now referred to as the `critical angle of attack' results in a loss of lift. This boundary condition would have been difficult (perhaps fatal!) to test without their wind tunnel. Similarly, virtual services are used to test scenarios that would be otherwise difficult to test. It can be difficult to elicit certain responses from a physical system, which you may want to do for functional testing. For example, how does my application react to a certain error condition? In addition to happy path testing, virtual services make it possible to test negative and boundary conditions with ease. Whilst it may be difficult to generate a certain response from a physical system, we can easily elicit any response we like from a virtual service (negative account balance, leap year condition, out of stock message etc.). This capability is especially useful if we are dealing with third party services, which are normally outside our control.
  - **Regression testing** – Consider the process flow below. In order to respond to a request, a system must query other downstream systems. If this system undergoes regular changes, we may need regular regression testing to ensure the integrity of existing functionality. A regression test will involve sending a number of test inputs (representing distinct test scenarios) and measuring the responses against a baseline. Any deltas against the baseline may represent broken functionality.



There are two challenges in creating this regression test rig. Firstly, we depend upon the downstream systems in order to run our test scenarios (and these may not always be available). Secondly, how will we know whether a delta in the response is caused by something broken in our system under test, ***or*** by something changed in one of the downstream

systems? By using virtual services in place of the downstream systems, we eliminate both of these challenges. The virtual services are permanently available to us. Furthermore, we remove uncertainty imposed by downstream systems. We can depend upon a virtual service to supply the same response to a given request, no matter how many times we call it. We therefore gain exactly what is needed for a controlled regression test rig – if a regression test results in an unexpected response, we know that our system under test is the only variable in the system, and must therefore be responsible for the delta.

- **Non-Functional Testing**
    - **Load testing** – For load testing, we typically have an end-to-end test environment configured specifically for the load test. We use LoadRunner, or a similar injection tool, to send lots of traffic to a front-end and measure response times back from the front-end. Provisioning that end-to-end environment is a big job! If the objective is to test production volumes, then the end-to-end infrastructure must be *at least as big as* the production environment in order to support the volumes. And how much test data is needed to support those volumes? Provisioning this hardware, software and test data is costly and labour intensive. The use of virtual services allows us to focus on the application of interest, and replace the downstream systems with virtual services. This negates the need for much hardware and test data, thereby greatly reducing the overhead associated with each load test. Further, this approach of focusing on the application of interest means that we can learn performance metrics for each component within the system (rather than the black-box traditional load testing approach). This allows us to decompose the SLA, learn about the behaviour of individual components, and identify potential bottlenecks.
    - **Performance testing** – We may change response times for virtual services. This allows us to test the impact of response times on our system in scope. From a performance testing perspective, this is interesting. How will our application react if response times from a downstream system increase? Will it degrade gracefully? Will session pools expand? Will we see out-of-memory errors? Playing with virtual services lets us find out. One real-life example – a UK based performance testing team in a large bank had to conduct their tests between the hours of 2-3pm each day. Why? During this time, a mainframe application in the US was under its highest load from end-users, and the testing team had to capture the resultant response times for their testing. The use case here for virtual services was obvious – use virtual services in place of the real mainframe, and elicit those response times *on demand.*

      There are similar use cases in retail, where seasonal peaks are a problem for every online retailer. Virtual services provide a means of testing .com

infrastructure against high watermark throughputs and beyond (worst case scenarios) in the safety of a non-production environment.

- **User Acceptance Testing** – As a general rule, virtual services should *not* be used for UAT. However, there are exceptions, such as a large online retailer preparing for the launch of the iPhone 4s. They knew that launch day would see a bombardment of online traffic through their dotcom site, and any outage or SLA violations would result in massive lost revenue. They made extensive use of virtual services throughout their SDLC to ensure that the dotcom infrastructure was up to the job. The week before launch, they conducted UAT, for which they switched off the virtual services, and used real systems instead. However, during that week, one of their Business-to-Business partners suffered an outage, which meant they could not complete business processes without this core service. In this case, they were able to continue UAT by reverting to the virtual service, which mimicked that partner. Without this virtual service, their iPhone 4s launch would have been jeopardised.

Creating Virtual Services

With Service Virtualisation, the objective is to create virtual services, which may be used in place of real systems. There are several means of creating the virtual services:

- **Recording –** in which the toolset *records* traffic between the in-scope system and a downstream out-of-scope system. Service Virtualisation tools can record traffic, regardless of the transport protocols (http/s, MQ, JMS etc.), message protocols (fixed width, csv, xml, SOAP etc.) and security protocols implemented at transport and/or message layers. A tool may be left in record mode for a period of time, in order to `learn' the behaviour of a system. Doing this allows the tool to create a virtual service, which `knows' how a real system responds to a variety of specific requests. A benefit of creating services in this manner is that the tool also learns about response times of the system.
- **Creation from message pairs –** Service Virtualisation tools can construct virtual services from request response message pairs (such as those taken from production logs). Some tools include the ability to desensitise messages (replacing customer names etc.).
- **Creation from meta data –** Service Virtualisation tools can construct barebones virtual services from WSDLs or COBOL copybooks. Obviously, a WSDL description only describes the signature of operations. A virtual service created using this method may be enhanced to include specific request/response behaviour. Service Virtualisation tools should make this a painless process. This method is used, for example, in SOAs where some systems are not yet available.
- **`Packet sniffing' –** using tools such as wireshark to `sniff' network traffic, and thereby learn the behaviour of a system.

With one of the benefits being efficiency, virtual services must be easy to create and maintain. Were it not quick and efficient, there would be little benefit over manual stubbing and mocking. Service Virtualisation tool sets therefore make a point of ensuring this creation is a simple process.

Conclusion

Service Virtualisation is a new field – existing practitioners are early adopters. In the near future, we should expect it to be the norm. The Wright brothers pioneered not just flight, but the practice of wind tunnels and models in aeronautical engineering. These afforded them advantages in terms of time-to-market and quality that ensured that the aeronautical industry embedded them as de facto practices. So too, early adopters of Service Virtualisation are experiencing similar benefits, which will set new standards in SDLCs.

References

1. Murer, S. *Testing At Large*. 2012 September. Speech presented at Software Engineering Dept, University of Oxford.
2. Software Engineering Economics. Prentice Hall. 1981. Barry W. Boehm.
3. http://vimeo.com/user7995852/review/30087515/b128c84133
4. Norton, D. *Agile Soup to Nuts – What Does A Truly Agile Organisation Look Like'.* 2013 May. Gartner Application Architecture, Development and Integration Summit, London.