# Microservices and Chronicle

This document discusses the popular architectural approach for modern systems known as *Microservices*, and how the Chronicle software suite supports this with a particular emphasis on low latency and maximum performance.

## Introduction - About Microservices

For some time now, software architecture practice has evolved away from building systems as large, self contained "monoliths" towards a more component focussed approach, via the SOA or "Service Oriented Architecture" to "Microservices". The idea behind this migration is that it allows minimising of coupling between the components of a system, while at the same time attempting to maximise the cohesion of the components themselves.

A more detailed definition of the term Microservices is provided by Martin Fowler here. However, we may consider the ideas to be a refinement of those in SOA. Microservices are more self-contained than the more traditional SOA services, adopting a "share nothing" approach wherever possible. While the name implies that they should be very small, in reality this means "small" in a functional sense rather than necessarily in the code-size sense. Microservices are intended to perform single tasks, in isolation, and to communicate with other microservices using well defined protocols.

We can think of this as the same approach taken by the designers of the Unix™ family of operating systems, especially with regard to the toolset, its focus on simple tools that perform individual tasks and their ability to connect together through the use of byte streams (either files or pipes) to pass data. This was documented as far back as July 1978 by Doug McIlroy in the foreword of the Bell System Technical Journal where Unix was first reported and has been refined and restated many times since.

However, as systems grow larger and more complex, the ability to manage large numbers of individual components introduces more and more complexity. Microservices by their nature are designed to exist in separate processes, perhaps distributed across separate machines. Communication between them becomes inter-process communication, which is significantly slower and more error prone than the simple "procedure-call" approach that is used in the traditional monolithic approach. One could argue that Microservices do not remove the complexity of monolithic systems but transfer that complexity to a distributed system which is potentially much more complex.

As a result of this, there is a reluctance among many organisations to adopt the Microservice approach in their systems architecture. However, when we look more closely, Microservices becomes more like a distillation of what are generally

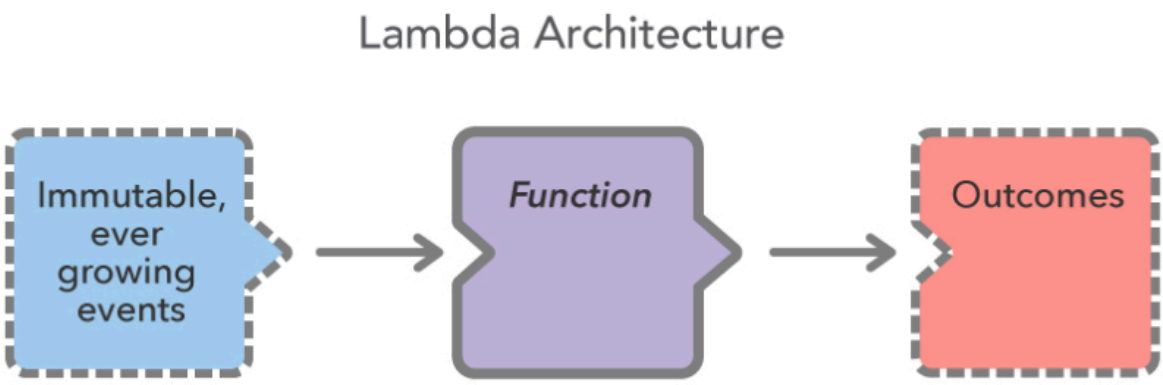
considered architecturally good practices, many of which may already be in use in a development team.

At Chronicle, our philosophy is based strongly on the use of Microservices in constructing sophisticated systems, especially in latency sensitive areas such as Financial trading systems. Chronicle Services has evolved to support this approach, providing the capability to implement low-latency, reliable, flexible and extensible microservice based software systems that have been successfully deployed with customers in the Financial Services Industry.

## The Chronicle Approach to Microservices

Systems built with Chronicle software are typically deployed into highly latency sensitive environments. Typical requirements are that a response to an event should be generated within 30 microseconds, with a 99% generated within 100 microseconds. Meeting these requirements requires careful attention to all aspects of coding and architecture.

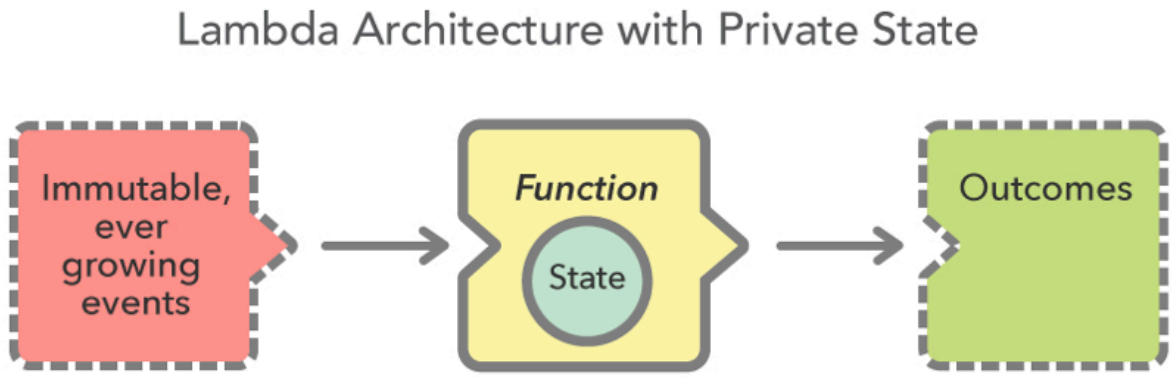
However, we have found that the Microservices architectural approach described above provides a good basis for building effective solutions that can be used in such demanding environments. We use the Lambda Architecture as a model for our view of Microservices:




Key aspects of this approach are that a function (Microservice) receives events from a source of data, reacts in some way to the events and generates output based on processing the input data. Lambda architecture is based on the idea that the data source for the function contains the complete history of all events, and operates in an append-only manner. Moreover, the function is stateless - any state information it requires is derived from its input event(s). This is a simple model, and can lead to efficient implementations.

However, it will not be suitable in every case. In particular for some functions it will be necessary to have state that is constructed from historical inputs, and not just from the current input. It is not practical to reconstruct this state on every input, so a slight variation of the approach can be used, where a function caches state locally.

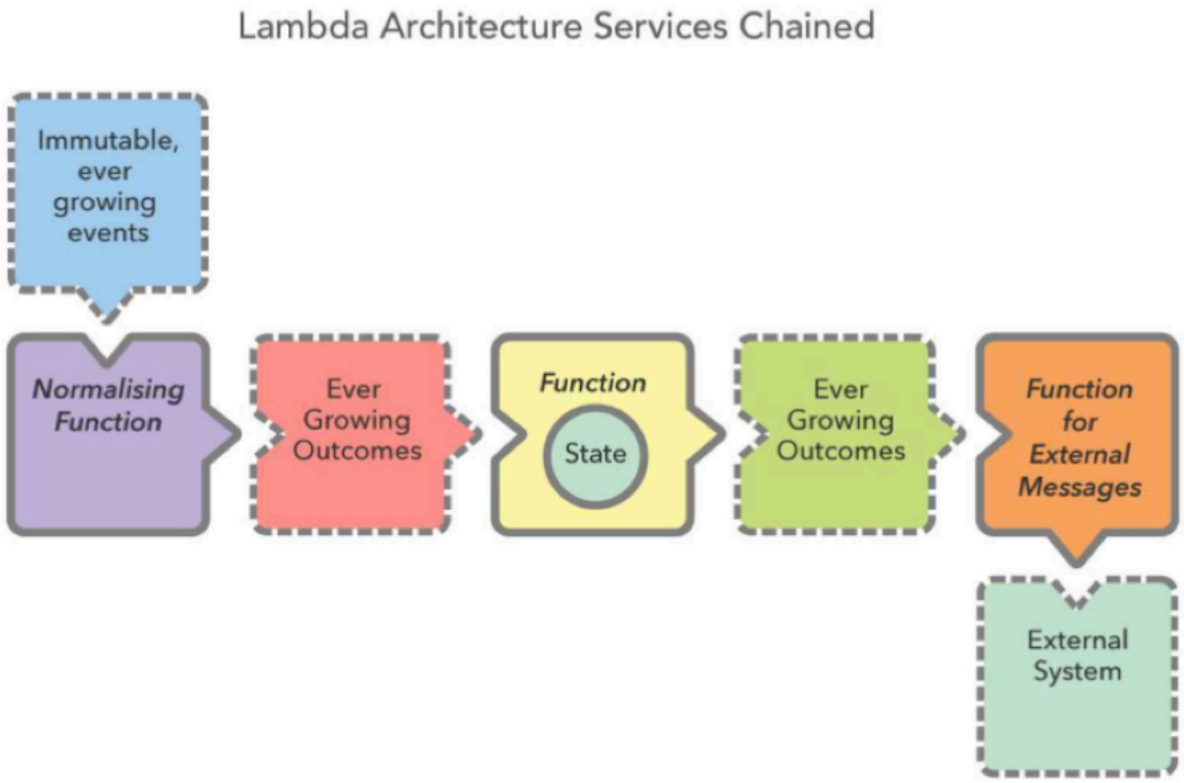
We do not expect the semantics of the function to change as a result of storing state - it is purely an optimisation, and at any point the state can be recreated from the inputs if necessary:




Transmission of events to, and from, a function in this way can be viewed as asynchronous messaging, which provides the most efficient mechanism for inter-service communication.

Notice that we do not mention any specific transport for these events. In the case where the sending and receiving services are part of the same process (i.e. running inside the same JVM) the transport can be as simple as a method invocation. However, in the more likely case where the services are in separate JVMs, some form of efficient cross-process transport is required. The interface from the services themselves to this transport should not mandate one of these over the other.

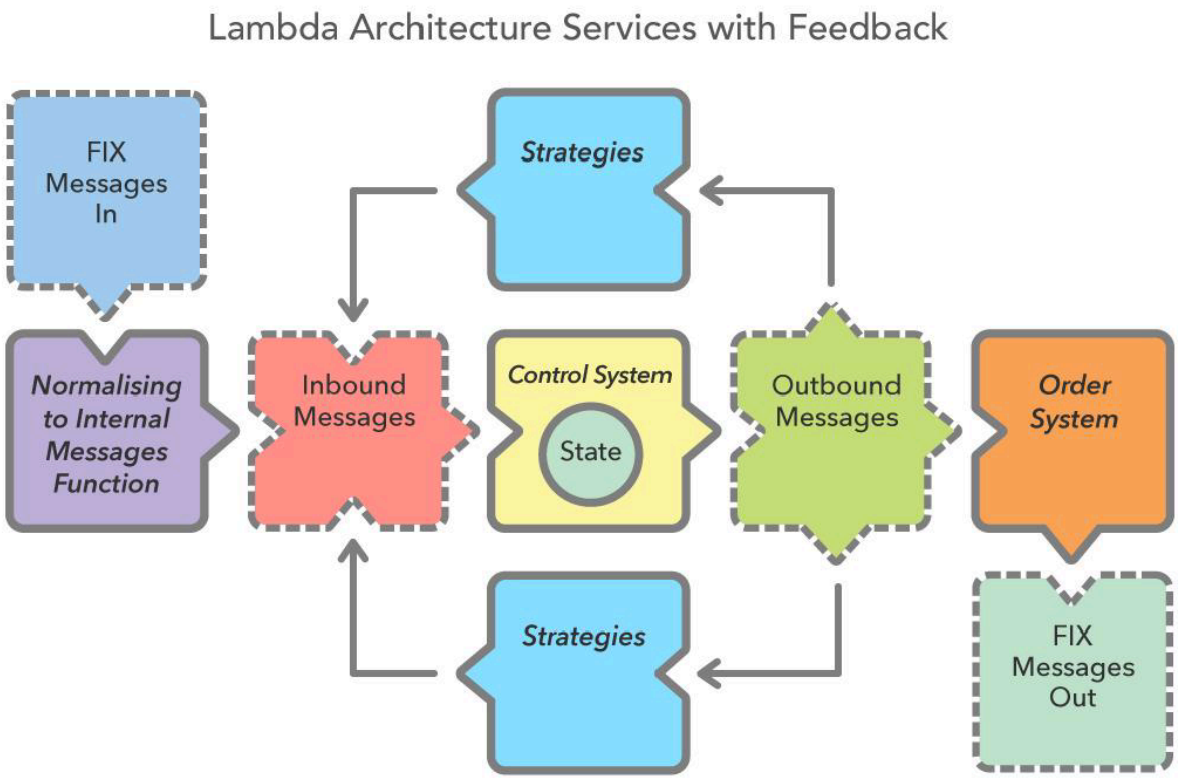
It should be clear that the above approach allows Chronicle microservices defined in this way to be chained together to produce more sophisticated processing pipelines, for example:




The "Normalising Function", and the "Function for External Messages" are examples of the simpler form of Lambda architecture. Their job is simple - to convert to and from an application specific canonical representation of the messages being sent to and from the system.

The main function processes requests as they appear in its input, which again is an example of an immutable collection of input messages. However the function is able to cache state constructed from these messages locally, allowing it to perform operations that are more complex than simple "one in one out" conversions. It will have access to historical data to aid in its calculations.

As this central functionality becomes yet more complex, it may involve sub-functions to enrich the data through accessing one or more databases, or other calculations such as the application of strategies that take some time to complete. To avoid blocking the main path of processing through the system, these potentially blocking operations can be factored out into separate services, each with the same underlying model:




This diagram shows a basic trading system that receives Market Data from exchanges as FIX messages, transforms them into an internal format and passes them to a central controlling function. From here the events are passed to one or more Strategy functions, each of which may produce a recommendation to place an order through a further Order Management system, which itself sends the orders to the exchanges using FIX messages. At all stages, the aim is to keep the individual Function components operating as efficiently as possible in order to minimise the latency between an inbound FIX message and any resulting order being places as a result of its analysis.

**Reproducibility and Message Storage in the Chronicle Approach**

The above approach to Microservices assumes that, at each stage, all messages that have passed through the stage are stored in perpetuity. This is key to being able to reproduce the data flow through the system, or some part of the system, at any time.

There are many advantages to this.

- Diagnosing and fixing bugs in any part of the system becomes easier, since the failing component can be examined over and over using the same data that identified the problem in the first place. When a fix is applied, testing using the same data can verify that the bug is no longer present
- Examining behaviour with the actual data being passed may identify algorithmic or data flow improvements, such as reducing the size of message payloads or removing redundant messages. This can be important in systems where latency must be minimised.

- Prospective performance improvements may be easily tested, and refined, using production class data.

Clearly this approach places some considerable demands on the memory architecture of the (Java) applications. The normal Java heap is not suitable for storing all such messages, since the size required is likely to be well above the available physical memory on the machine where the application is running.

When a Java application heap size exceeds the amount of available physical memory there are seriously detrimental effects on the performance of the system. The host operating system has to page out portions of the heap to secondary storage, bringing them back into main memory when they are required again. The operating system has no knowledge of the layout of Java memory inside the heap and bases its decisions on which pages to evacuate to secondary storage solely on measures of when the page was last accessed.

The cost of garbage collection significantly increases as heap size grows to very large values, partly because of the sheer amount of work that must be performed during a collection, exacerbated by the costs of paging described above.

A solution is to use memory mapped files. These increase the amount of virtual memory that is available to a running Java application without affecting the size or operation of the garbage collected heap. The increase in available virtual memory does come at a cost, but this is relatively small in comparison to the benefit of having so much more memory available to the application.

Chronicle provides this functionality through Chronicle Queue. Chronicle Queue is open source software, whose efficiency and reliability has led to it being demonstrated as suitable for use in Microservices environments where latency in message passing is an important factor.

Chronicle Queue overlays data structures on the mapped files, allowing transparent access to the data in them from the application. Additionally, since files can be mapped into multiple processes, concurrent access to Queues from multiple JVMs (Microservices) is supported.

One of the reasons that mapped files provide high performance in this area is that changes to the data in the process are not directly written back to the file. The mapped files act as a *write-back* cache of the data on disk. This could result in a delay, typically around 50 microseconds per message, before the message is persisted to the disk file and presents a potential vulnerability in achieving the goal of guaranteeing that all messages are stored in perpetuity.

Chronicle avoids this by replicating a Queue to another JVM on a separate machine rather than relying on synchronising the messages to disk. In fact, the time taken to replicate can be significantly lower (less than 10 microseconds) than the time to store to even a SSD. Replication is a key feature of Chronicle Services and will be discussed in more detail later in this guide.

The sizes of the mapped files used by Chronicle Queues are such that Flow Control is not considered an issue. The files act as buffers between the components that are large enough to store enough data flow control is not needed, since there will always be enough space to store messages being written by a producer, even if the consumer is considerably slower in reading the messages. In the extreme case we can do without a consumer altogether, making these files into logs written by software components.

https://github.com/ChronicleEnterprise/Chronicle-Services/blob/master/docs/UserGuide/01Microservices.adoc