<https://dzone.com/articles/what-latency-throughput-and>

# What is Latency, Throughput and Degree of Concurrency?

**How do you define throughput and latency for your test?**

There is not a simple question, so I have replied with a post.

**Sustained Throughput**

I consider throughput to be the number of actions a process can perform over a sustained period of time, between 10 seconds and day. (Assuming you have a quite period over night to catch up) I measure this as the number of actions per second or mega-bytes (MB) per second, but I feel the test needs to run for more than a second to be robust. Shorter tests can still report a throughput of X/s but this can be unrealistic because systems are designed to handle bursts of actively with caches and buffers.

If you test one behaviour alone you get a figure which assumes nothing else is running on the system and the limits of these buffers are not important. When you run a real application on a real machine doing other things, they will not have full use of the caches, buffers, memory and bandwidth and you may not get within 2-3x the sustained throughput let alone the more optimistic burst throughput. A SATA HDD can report a burst throughput of 500 MB/second, but it might only achieve a sustained 40 MB/s. When running a real program you might expect to get 15-25 MB/sec.

**Latency**

There are two way to report latency. One way latency and round trip latency (or Round Trip Time). Often the first is reported because it is less, but it difficult to measure accurately as you need a synchronised clock at both ends. For this reason you often measure the round trip latency (as you can use just one accurate clock) and possibly halve it to infer the one way latency. I tend to be interested in what you can expect from a real application and the higher round trip latency is usually a better indication.

A common measure of latency is to take the inverse of the throughput. While this is easier to calculate, it is only comparable to other tests measured this way because it only gives you the most optimistic view of the latency. e.g. if you send messages asynchronously over TCP on loop back you may be able to send two million messages per second and you might infer that the latency is the inverse of 500 ns each. If you place a time stamp in each message you may find the typical time between sending a receiving is actually closer to 20 micro-seconds. What can you infer from this discrepancy? That there around 40 (20 us / 500 ns) messages in flight at any time.

**Typical, Average and Percentile Latency**

Typical latency can be calculated by taking the individual latencies, sorting them and taking the middle value. This can be a fairly optimistic value but because its the lowest, it can the value you might like to report. The Average latency is the sum of latencies divided by the count. This is often reported because its the simplest to calculate and understand which it means. Because it takes into account all values it can be more realistic than the typical latency. A more conservative view is to report a percentile of latency like 90%, 99%, 99.9% or even 99.99% latency. This is calculated by sorting the individual latencies and taking the highest 10%, 1%, 0.1% or 0.01%. As this represents the latency you will get most of the time, it is a better figure to work with. The typical latency is actually the 50% percentile.

It can be useful to compare the typical and average latencies to see how "flat" the distribution is. If the typical and average latencies are within 10%, I consider this to be fairly flat. Must higher than this indicates opportunities to optimise your performance. In a well performing system I look for about a factor of 2x in latency between the 90%, 99% and 99.9%. The distribution of Latencies often have what is called "fat tails". Every so often you will have values which are much larger than all the other values. These can be 10 - 1000x higher. This is what looking at the average or percentile latencies more important as these are the one which will cause you trouble. The typical latency is more useful for determining if the system can be optimised.

**A test which reports these latencies and throughputs**

The test "[How much difference can thread affinity make](http://vanillajava.blogspot.co.uk/2012/02/how-much-difference-can-thread-affinity.html)" is what I call an echo or ping test. One thread or process sends a short message which contains a timestamp. The service picks up the message and sends it back. The original sender reads the message and compares the timestamp in the message with another timestamp it takes when the message is read. The difference is the latency measured in nano-second (or micro-seconds in some tests I do)

**Wouldn't less latency lead to more throughput? Can you explain that concept in mere mortal terms?**

There are many techniques which improve both latency and throughput. e.g. using faster hardware, optimising the code to make it faster. However, some techniques improve only throughput OR latency. e.g. using buffering, batching or asynchronous communication (in NIO2) improves throughput, but at the cost of latency. Conversely making the code as simple as possible and reducing the number of hops tends to reduce latency but may not give as high throughput. e.g. send one byte at a time instead of using a Buffered stream. Each byte can be received with lower latency but throughput suffers.

**Can you explain that concept in mere mortal terms?**

In simplest terms, latency is the time per action and throughput is the number of actions per time. The other concept I use is the quantity "in flight" or "degree of concurrency", which is the *Concurrency = Throughput \* Latency*.

**Degree of Concurrency examples**

If a task takes 1 milli-second and the throughput is 1,000 per second, the degree of concurrency is 1 (1/1000 \* 1000). In other words the task is single threaded.  
If a task takes 20 micro-seconds and the throughput is 2 million messages per second, the number "in flight" is 40 (2e6 \* 20e-6)  
If a HDD has a latency of 8 ms but can write 40 MB/s, the amount of data written per seek is about 320 KB (40e6 B/s \* 8e-3 s = 3.2e5 B)

<https://medium.com/@malith.jayasinghe/performance-characteristics-of-non-blocking-systems-how-does-the-number-of-threads-impact-the-9926752595d1>

# Design of non-blocking systems: How to improve the tail latencies and load average by optimizing the number of threads?

[Malith Jayasinghe](https://medium.com/@malith.jayasinghe?source=post_page-----9926752595d1--------------------------------)

[Malith Jayasinghe](https://medium.com/@malith.jayasinghe?source=post_page-----9926752595d1--------------------------------)

[Apr 8, 2019·8 min read](https://medium.com/@malith.jayasinghe/performance-characteristics-of-non-blocking-systems-how-does-the-number-of-threads-impact-the-9926752595d1?source=post_page-----9926752595d1--------------------------------)

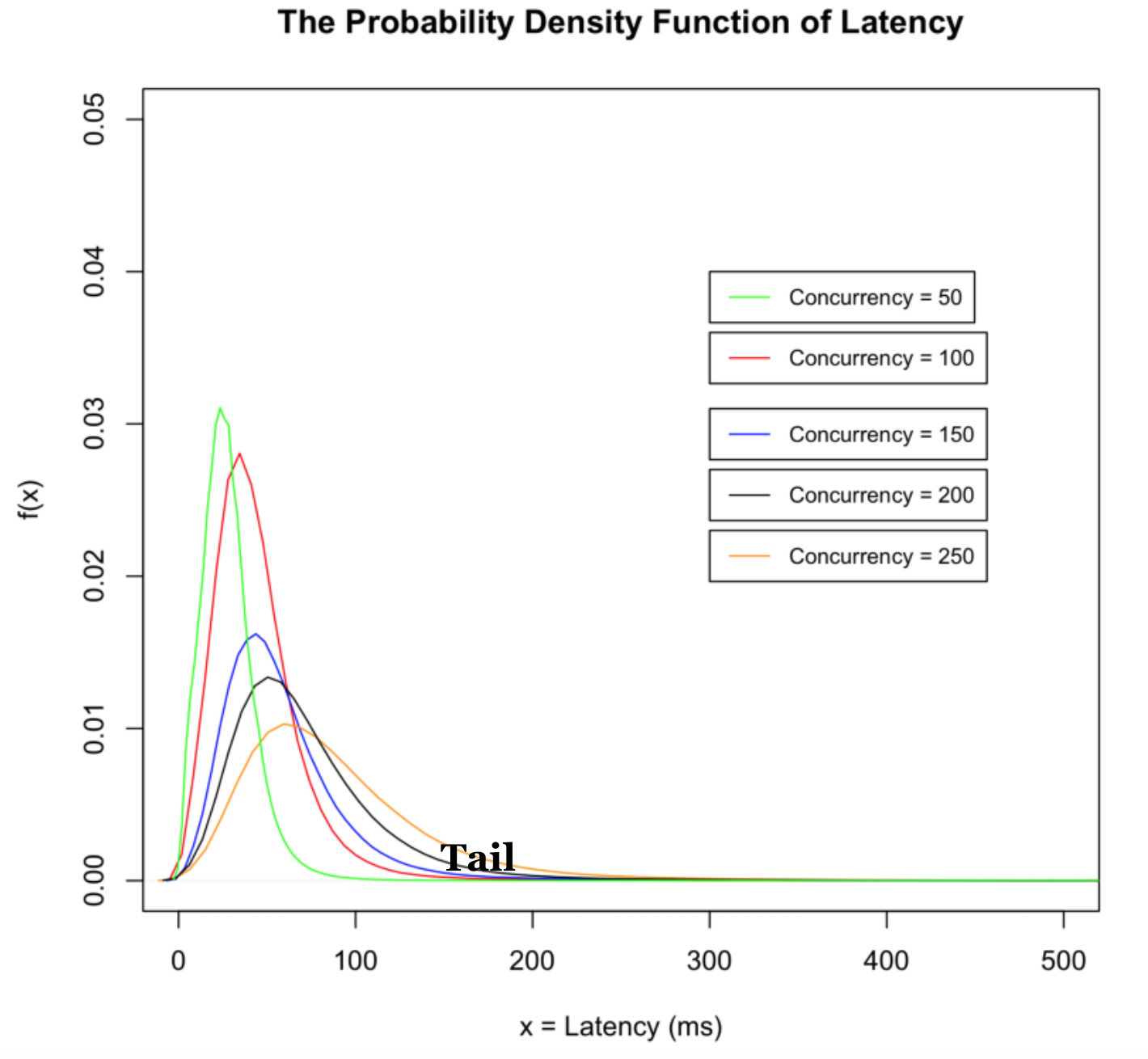
# **Introduction**

When a program executes, it carries out various types of activities such as computations, doing IO, waiting on a condition. At the execution time, these activities have to be mapped to OS-level threads, and we call this mapping the thread model of the language. Most programming languages use the blocking thread model. In a blocking thread model, when the program carries out a blocking action such as IO, the OS level thread also blocks. In contrast, a non-blocking system does not block an OS thread when the thread needs to block on a blocking operation (e.g. I/O) rather it frees up the OS thread. A blocking system, on the other hand, blocks the processing thread until the task is run to completion. On a blocking system, if we increase the number of OS threads available while increasing the load, throughput increases. However, at a certain point, throughput starts to degrade and latencies start to suffer. The tail latencies increase dramatically. This is because, as the number of OS threads in the system increases, the context switches overhead start to dominate and each request have to wait longer to be scheduled. In a blocking system, if all programs do only computations, the optimal number of OS threads would be the same as the number of cores. However, this does not work if threads also perform IO as IO could take significantly more time compared to CPU operations and in such cases, the CPU will idle while most threads waiting for IO. Non-blocking systems give us the best of both worlds by letting us do IO while keeping a small number of OS threads. As a result, the nonblocking model can provide better throughput and latency. Most programming languages only support a blocking model directly, and to build a nonblocking system, the programmer has to write clever code (e.g. using non-blocking IO).

On the other hand, programming languages such as Ballerina supports non-blocking models out of the box. [Ballerina](https://ballerina.io/) is a concurrent and strongly typed programming language optimized for integration.[**Ballerina**](https://ballerina.io/)**provides a great higher level of abstraction while hiding the complexity of the code underneath. Its execution model is composed of lightweight parallel worker units that are non-blocking where no function can lock an executing thread manifesting sequence concurrency.** **These lightweight parallel workers are backed by a thread pool with a fixed number of threads (scheduler threads) where each thread gets mapped to an OS thread. Since the workers of**[**Ballerina**](https://ballerina.io/)**do not block we can maintain the size of this thread pool at a minimal value (i.e. close to the number of cores). The objective of this article is to investigate how the size of this thread pool impacts the performance.**

# **Measuring the performance**

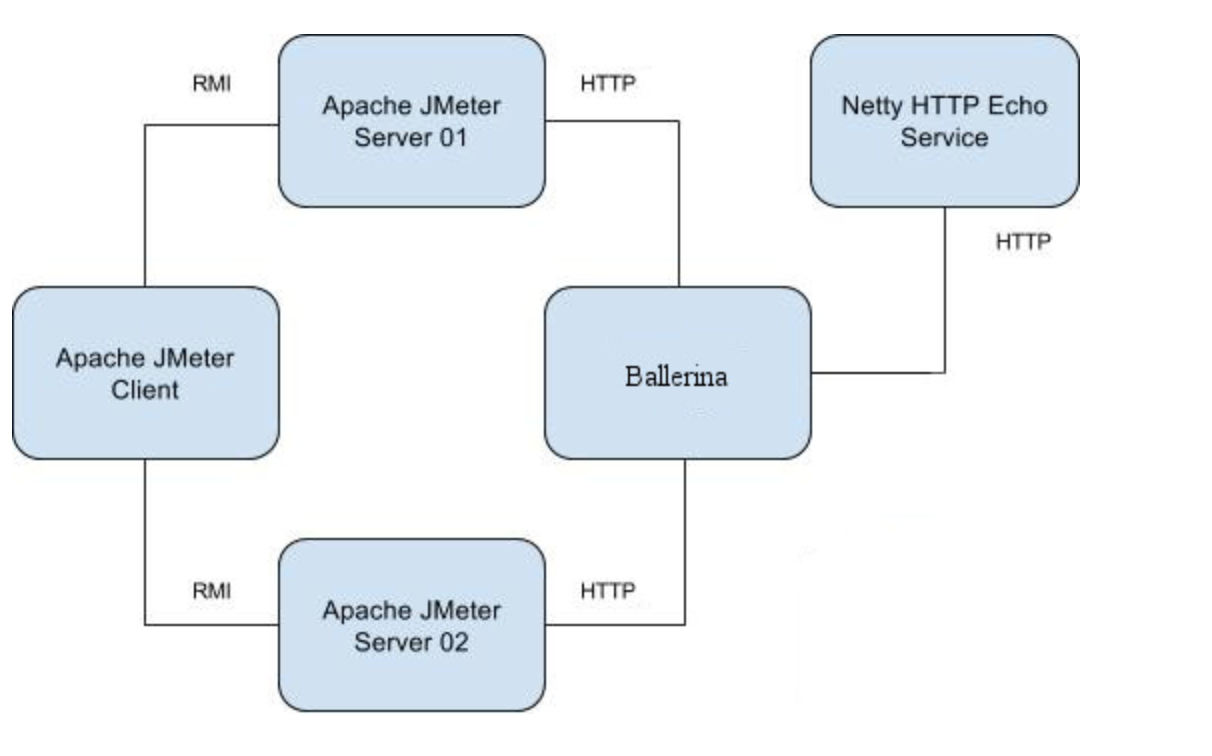
Let us provide some details of the performance metrics that we use to evaluate the performance. There are two main metrics: throughput and latency. The throughput measures the rate at which the system performs the work. The latency is a measure of waiting. If we consider a client-server system, the latency (of a request) is the total round-trip time. The latency is not a single value rather a set of values which represent individual latencies of many requests. In other words, the latency exists in the form of a distribution. The question is how do we analyze the latency if it exists in the form of distribution? One option is to use the average. However, there are issues with just using the average for measuring the [latency](https://www.loggly.com/blog/average-poor-metric-measuring-application-performance/). Therefore, when we analyze the latency, we need to consider the (full) distribution of latencies. Let us now try to understand what this means in detail. The following figure shows the latency distribution of a system under different load conditions (note: higher numbers of concurrent users represent higher load)



Note the behavior of the tail of the distribution under different concurrency levels (concurrent users). The tail of the distribution represents long latencies. Hence we use the term “tail-latencies” to describe long latencies. A distribution with **thicker and longer tails** will have a higher probability of long latencies from occurring. In the above figure, we see an increase in the tail latencies with the increasing number of concurrent users. The tail-latencies are a major concern in high-performing concurrent systems. As such, system designers are continuously making efforts to reduce the tail latencies of systems. The most (basic) metric that we can use to get an idea about the tail latencies is the latency [percentiles](https://en.wikipedia.org/wiki/Percentile). There are other types of analysis we can do to analyze the tail of the distributions (e.g. tail index).

# **Performance Test Details**

Let us now provide some details about the performance testing environment. The diagram below shows the deployment diagram.



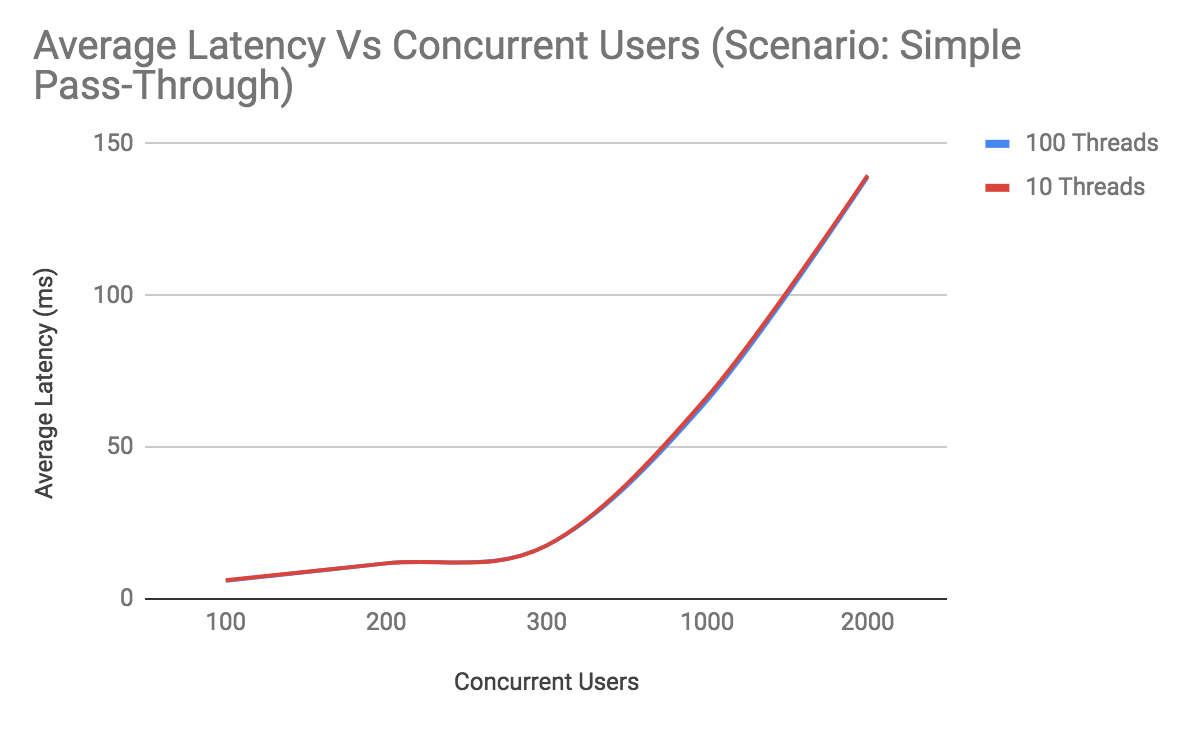
We ran the performance tests on Amazon EC2 using a C4-Xlarge instance for [Ballerina](https://ballerina.io/)(ballerina-0.971.1) and C3-Xlarge instances for the workload generator (3 instances) and the back-service.

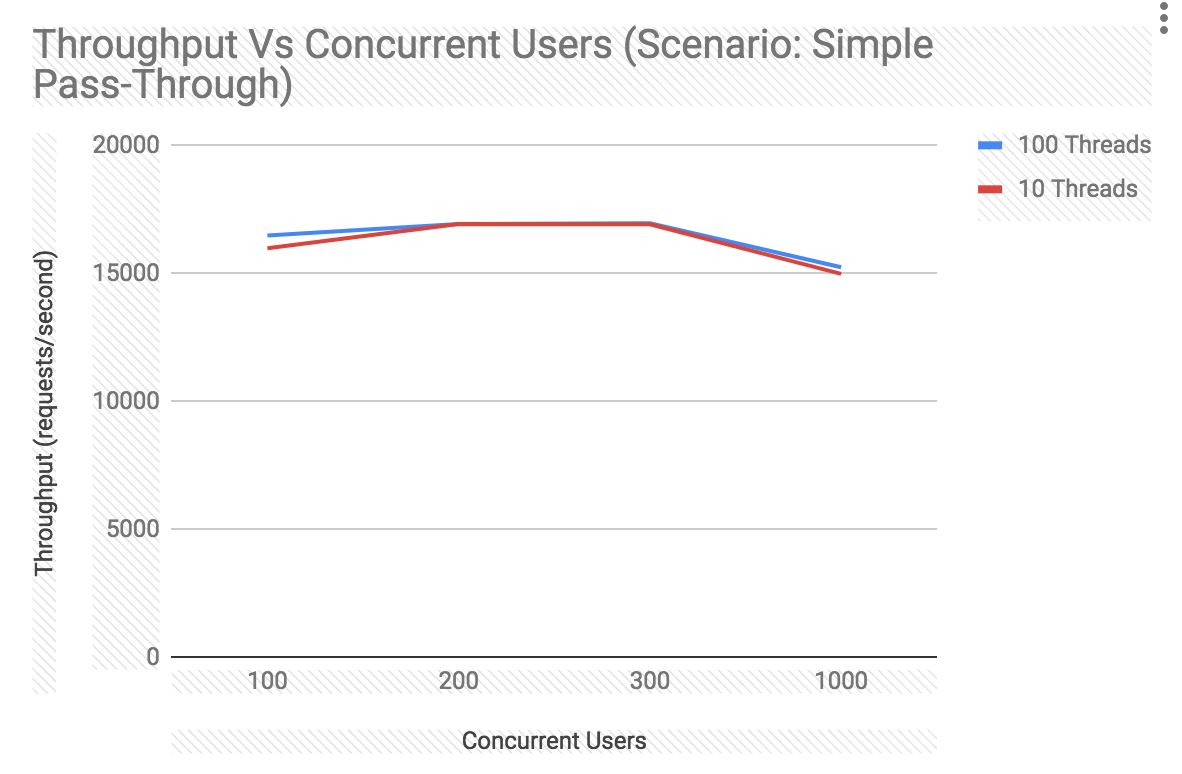
# **Performance Results**

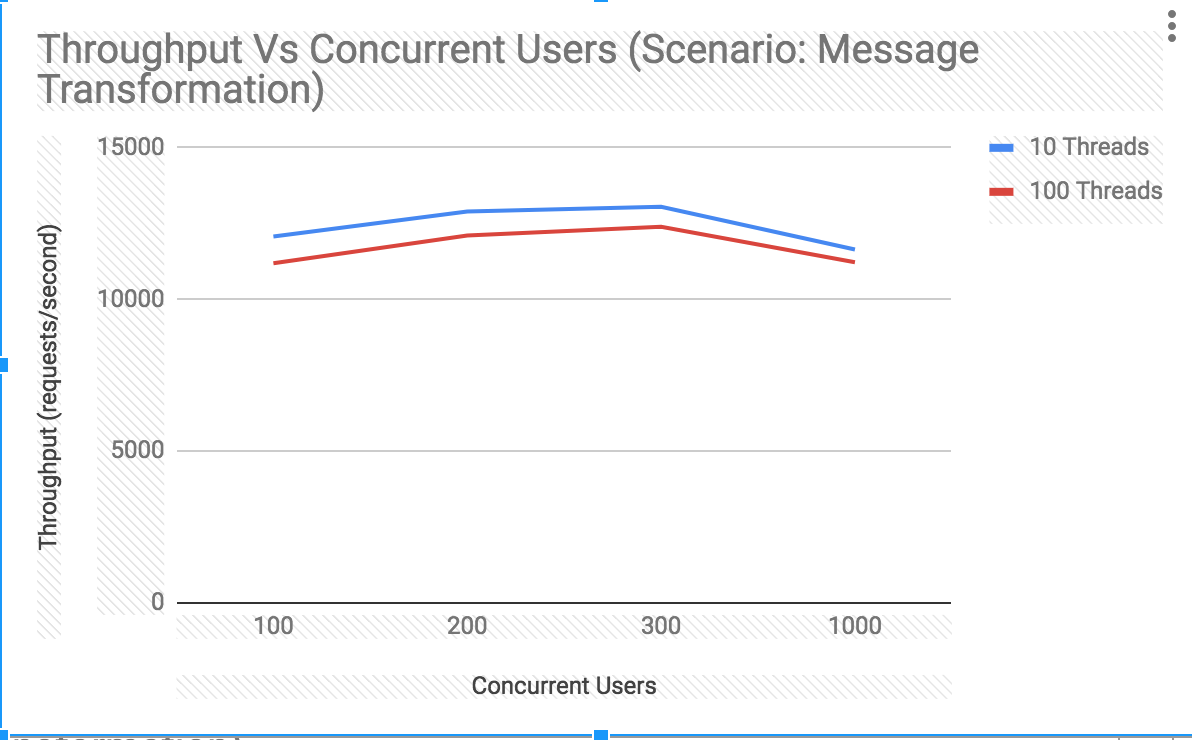
Let us now present the results for two use cases (1) simple pass-through and )(2) transformation. Under the simple pass-through scenario, the workload generator publishes the requests to the [Ballerina](https://ballerina.io/) service and the [Ballerina](https://ballerina.io/) service posts these requests to a back-end service, the back-end service echoes back the requests posted to it. The difference between simple pass-through and transformation is that under the transformation use case [Ballerina](https://ballerina.io/) service performs a message transformation in the middle of message flow. As already pointed out, [Ballerina](https://ballerina.io/) has a non-blocking architecture. Its concurrency model consists of lightweight workers backed by a thread pool with a fixed number of threads (processing threads) where each thread gets mapped to OS thread. Whenever there is a blocking operation (network I/O in this case) the workers go into a non-active mode and as a result, the processing thread never gets blocked.

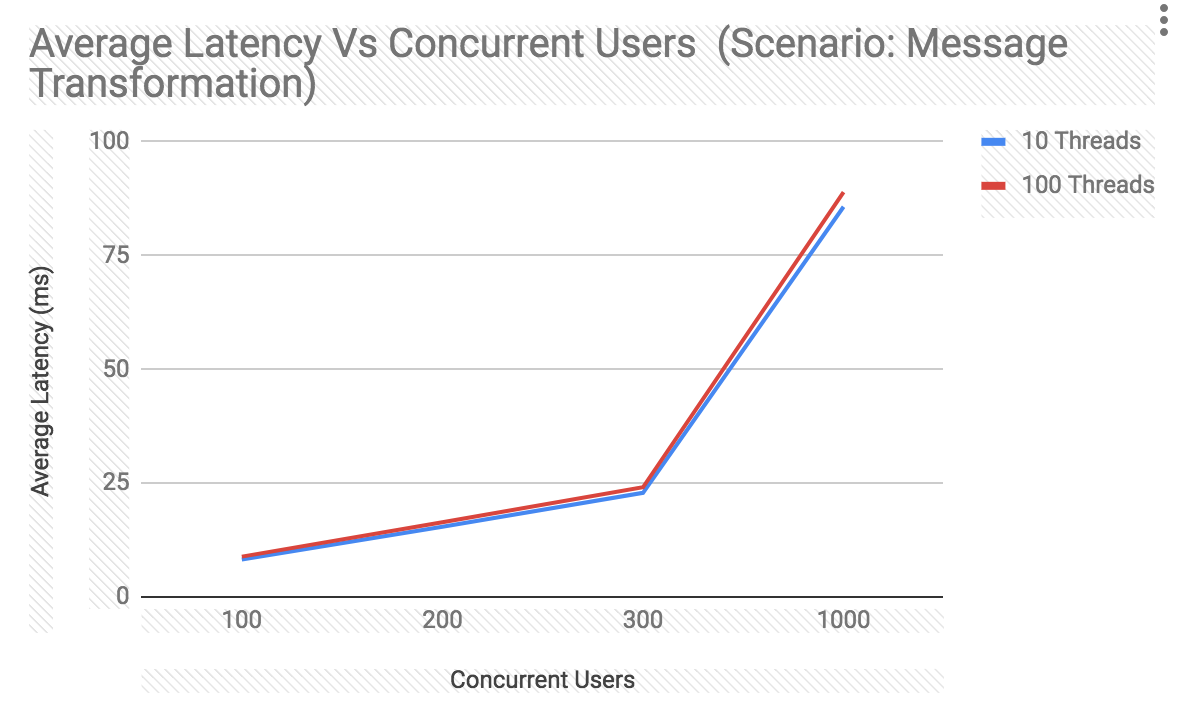
## The Impact of average latency and throughput

The following figures show how the throughput and average latency vary with the number of concurrent users (message size used 50 B) when we use 10 threads and 100 threads in the processing thread pool





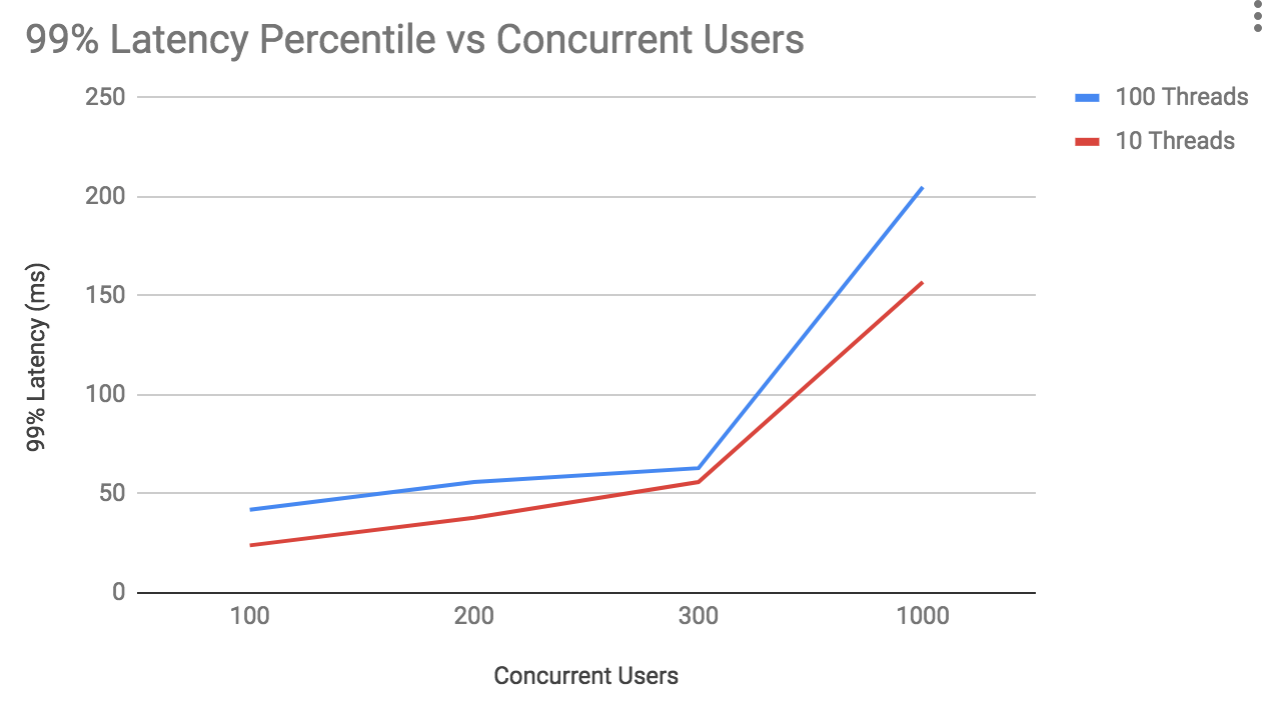




We noticed that there is no significant difference in the average latency whether we use 10 or 100 threads in the pool. There is a minor improvement in the TPS when using 10 threads for the transformation scenario. The maximum improvement in TPS is about 7%. T**he important observation here is that just using 10 threads the system can serve a large number of users while achieving very high throughput (up to 17000 requests/second)**. This is because the threads never block when they perform IO (network I/O in this case).

## The impact of tail latencies

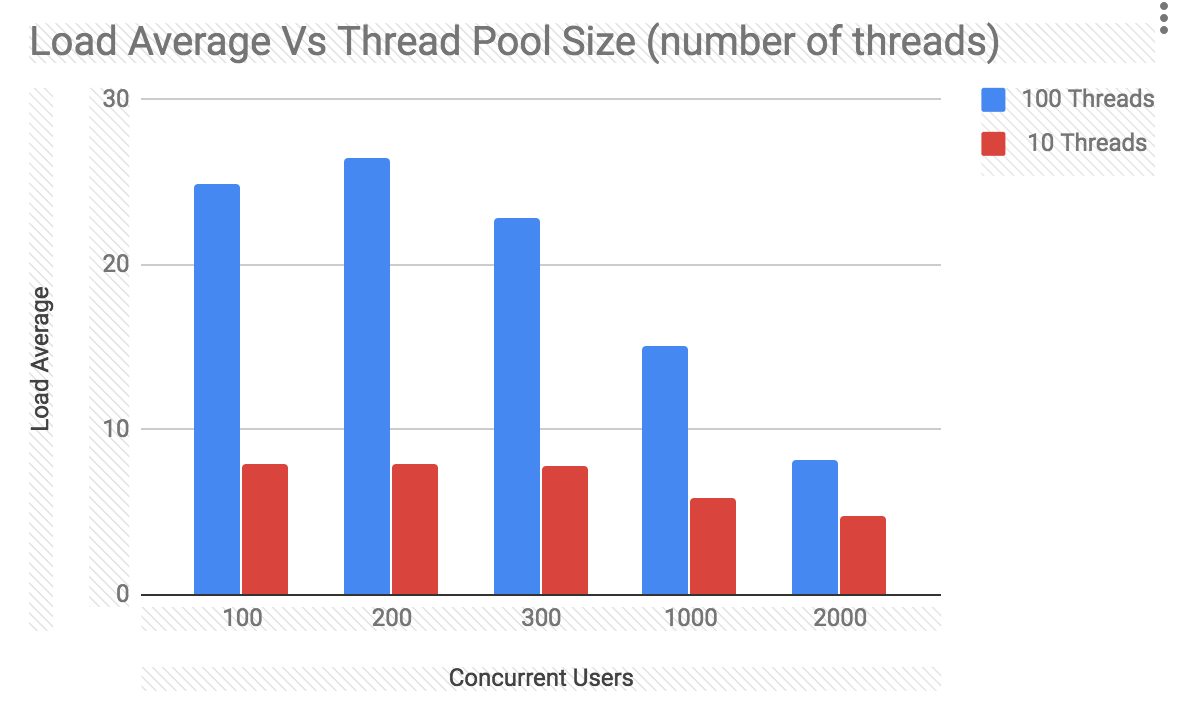
Let us now investigate the impact of the number of threads on the tail latencies. The following figure shows the behavior of 99% latency percentiles when using 100 and 10 threads in the pool (scenario message transformation, message size 50B).



We do notice that as we increase the number of threads, it increases the occurrence of long-tail latencies. The way in which the number of threads impacts the tail latencies depends on the use case. For the I/O bound use cases (i.e. simple pass-through) we did not notice a significant degradation on the tail latencies with the increasing numbers of threads. However, as the workload becomes more and more CPU bound, having a large number of threads, increases the probability of long-tail latencies from occurring (indicated by higher latency percentile values). We note that 100 threads have higher 99% percentile values compared to that of 10 threads. One possible reason for the increase in the tail latencies under a large number of threads is the increased context switch overhead.

## The impact of load average

Let us now discuss the impact of the number of threads on the load average. Our observation is that when using a large number of threads, there is a significant increase in the load average in particular for workloads that have both I/O and CPU operations. The following figure shows the behaviour for message transformation use case.



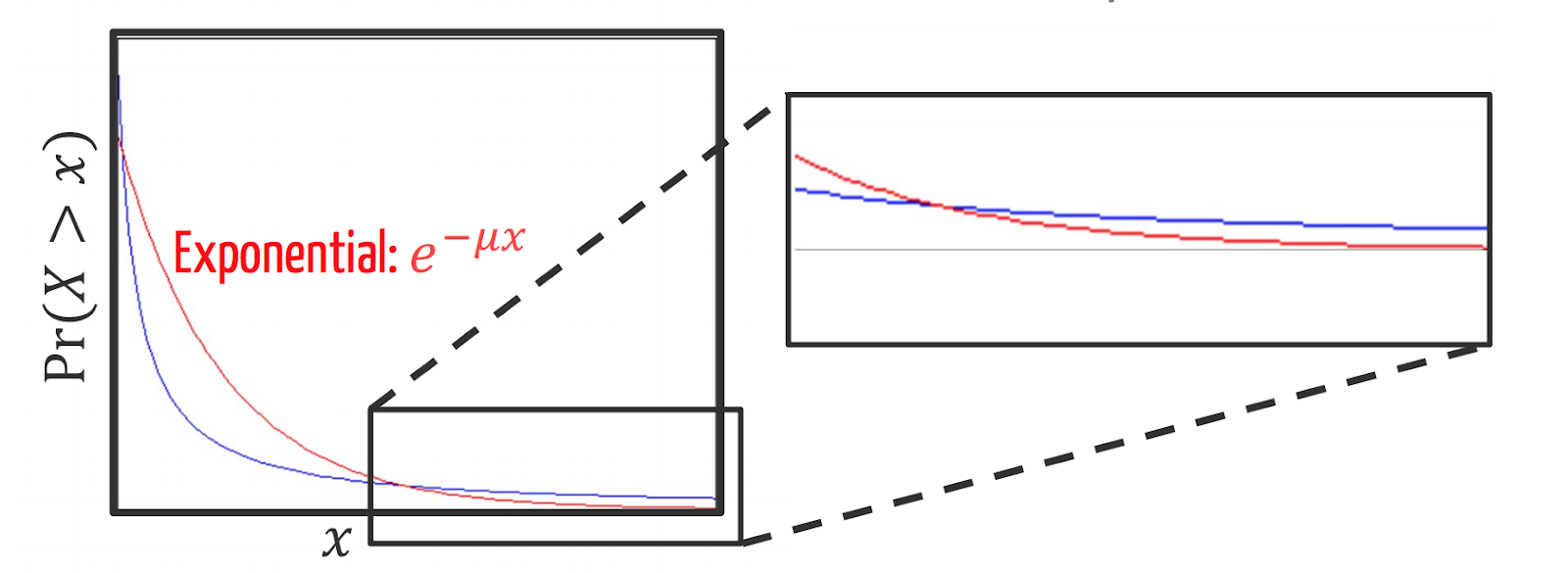
Let us now try to understand why we get high load average when the number of threads is high. There are a number of factors which contributes to the load average and one main one is the CPU run-queue length (OS level queue which contains tasks queued waiting to run). Let’s assume that we have 4 threads and this means that the server will process a maximum of up to 4 requests simultaneously at a given point in time. If a new request arrives while these requests are being served, the request will not be processed immediately rather it will be queued in an application level queue (not in a run-queue). As a result, there is less queueing at the OS level (i.e. run-queue) which leads to a lower load average. Numbers obtained for 4 and 100 threads are shown below.

4 threads: run-queue length = 8, load average = 5

100 threads: run-queue length 41, load average = 37

# **Advanced methods for analyzing latency distributions**

The average, percentiles, maximum and standard deviation are the main metrics that are used in the industry when analyzing the latency numbers of systems. In addition to these, we can use advanced statistical methods to further analyze and understand properties of latency distributions.For example, we can try to characterize the tail of a latency distribution and see if the distribution is a heavy-tail or long-tail. A distribution with a long-tail has a heavier tail compared to an exponential distribution. This is illustrated in the following figure (heavy-tail distribution in blue, exponential distribution in red).



[Source](http://users.cms.caltech.edu/~adamw/papers/2013-SIGMETRICS-heavytails.pdf)

We have analyzed and identified the tail characteristics of different types of systems and I will share our results in our future blog posts and research papers.

# **Conclusion**

In this blog post, we looked at the performance characteristics of non-blocking systems. We implemented our use cases using [Ballerina](https://ballerina.io/), a programming language designed for the integration of systems.[Ballerina](https://ballerina.io/) has a non-blocking architecture. Our particular focus was to investigate how the number of processing threads impacts the performance. We noticed that a non-blocking system was able to handle a large number of concurrent users while achieving higher throughput and lower latency with a small number of threads. We then looked at how the number of processing threads impacts the performance. We noticed a minimal impact on throughput and average latency on the number of threads. However, as the number of threads increases, we see a significant increase in the tail latencies (i.e. latency percentiles) and load average.