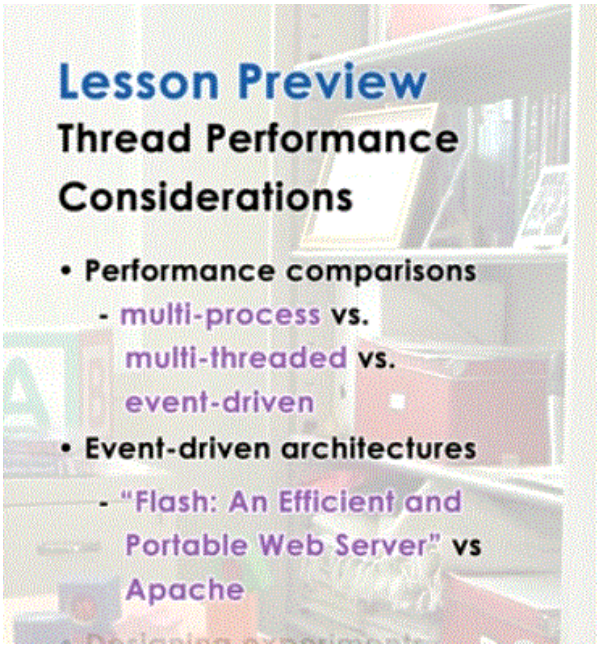
P2L5 - Thread Performance Considerations

01 - Lesson Preview



In this lecture, we will contrast several approaches for structuring applications that require concurrency. This will include a comparison between multi-process and multi-threaded approaches and we talked about both of these already in this course. And in addition, we will present an alternative, a so called event-driven approach.

We will base the discussion of the event driven model on Vivek Pai's paper "Flash: An Efficient and Portable Web Server". This paper describes the event driven architecture and it also includes detailed performance comparisons between a multi-process, multi-threaded, and event-driven implementation of a web server application.

In addition, these results are put in contrast with Apache, which is a popular open source web server. We will end this lecture with a more generic discussion on how to structure good experiments.

02 - Which Threading Model is Better

During the threads and concurrency lecture, recall that we had a quiz in which we were comparing the Boss-Worker model with the Pipeline model. And we did that specifically, in one example, for 6 worker threads, in both cases, and for a toy order that consisted of 11 toys. For the Boss-Worker model we said that takes 120 ms (milliseconds) for a worker to process a toy order, and then for the Pipeline model we said that it takes 20 ms to complete each of the pipeline stages and a full toy order had 6 pipeline stages.

Let's compare these two models to see which one is better. The computation that we performed during that homework, showed us that regarding execution time the Boss-Worker model took 360 ms for the 11 toy orders, and the Pipeline model 320 ms for the same 11 toy orders.

Now let's consider something else, now let's compare these two models with respect to the average time they take to complete an order. To find the average time, we have to sum up the times that it took to complete every single one of the 11 orders and then divide by 11. The first 5 orders took 120 ms to complete; they were executed by the first group of 5 threads. The second 5 orders were scheduled in the second batch so they took twice as long, they took 240 ms to complete. And then the 11th toy order took 360 ms to complete; it had to wait, it could only be

started until the previous 10 orders were completed in the groups of 5 plus 5 threads each. So if we compute this, the average time to complete an order for the Boss-Workers model is 196 ms.

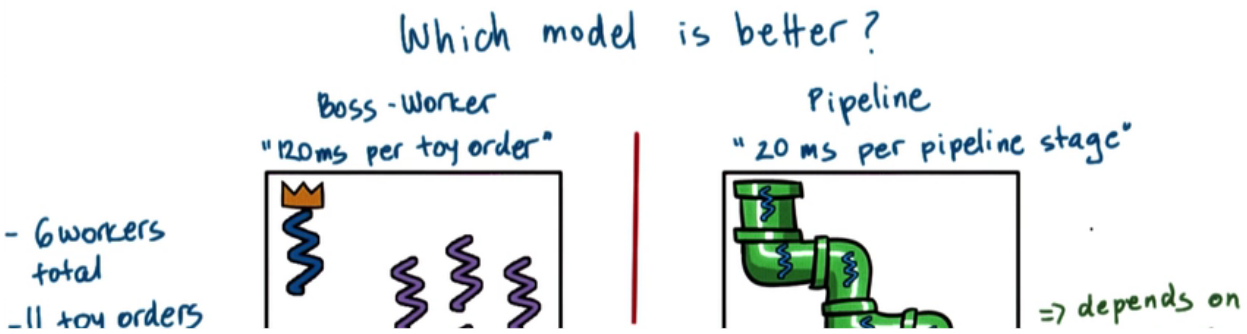


Image Errata: The image shows 6 worker threads but the calculations are done for only 5 worker threads. Also, In Boss-Worker model ((5*120) + (5*240) * 360)/11

should be ((5*120) + (5*240) + 360)/11

If we take a look at the Pipeline model, the first order took 120 ms to complete. 6 pipeline stages times 20 ms. The next one was already in the pipeline and once the first order completed, it had to finish the last stage of the pipeline, so its completion time will be 20 ms longer, so 140 ms. The one that came after that, another 20 ms longer for 160 ms, and so on until the very last order which will take 320 ms. So the average completion time for the Pipeline model is 220 ms.

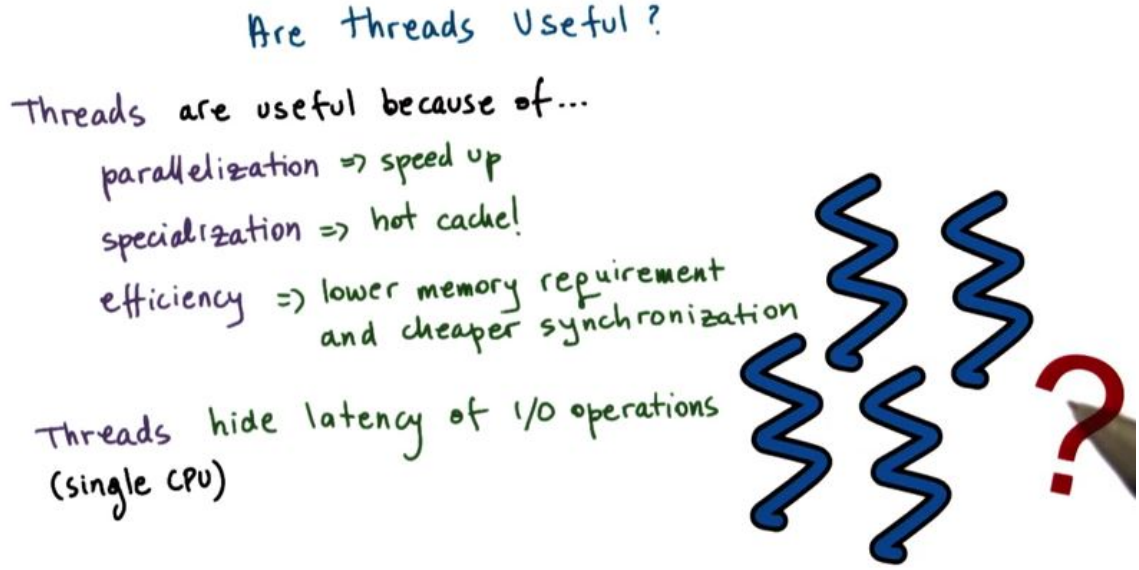
So basically what this shows us is that if we consider execution time, if that's what's important, then we should pick for this particular configuration for 11 toy orders and 6 workers, then we should pick the Pipeline model, its better, it leads to shorter execution time. However, if what we care for is average time to complete the order, because that's what makes our customers happy, then the Boss-Worker model is better, its average completion time is 196 ms compared to 220 ms for the exact same workload, 11 toy orders, and the exact same number of workers working in the toy shop. If you slightly modify this problem, where you look at a situation with a different number of workers or different number of orders then you may end up coming up with completely different answers and drawing different conclusions as to which one of these models is better.

Now the important thing, is though that when we look at a specific configuration, say we are the toyshop manager who is concerned with completing as many orders as possible in a fixed

amount of time, then maybe we will choose, under these circumstances, the Pipeline model because it gives us shorter completion time. However, if we are the customers of that toy store, then we probably would prefer if this (Boss-Worker) was the model that the toyshop implemented because the orders will be completed in a shorter amount of time. An important conclusion of this, is that when we're comparing two different implementations of a particular problem, or two different solution design points, then its very important to think about what are the ***metrics*** that are used to evaluate those different solutions, those different implementations. Who cares about that?

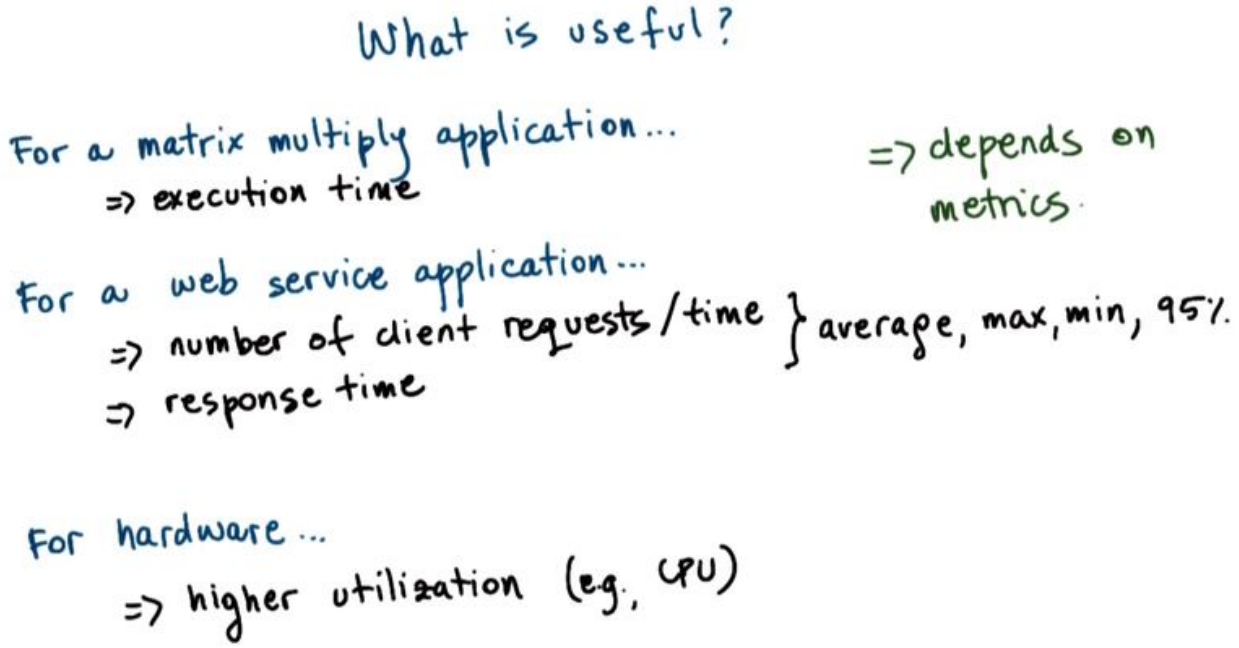
03 - Are Threads Useful

At the start of the threads and concurrency lecture, we asked ourselves whether threads are useful and we mentioned there are a number of reasons why they are. They allow us to gain speed up because we can parallelize problems, they allow us to benefit from a hot cache because we can specialize what a particular thread is doing on a given CPU, and they lead to implementations that have lower memory requirements (efficiency) and where its cheaper to synchronize compared to multi-process implementations of the same problem. We said that threads are useful even on a single CPU because they let us hide the latency of I/O operations. However, how do we draw these conclusions? What were the workloads, what were the kinds of resources that were available in the system, and ultimately what were the different metrics that we were using when comparing different implementations with and without threads.



And the way that we would measure whether something is useful or not would differ. For instance, for a matrix multiply application we want to think about the execution time of an implementation, or a solution. Or for a web service application maybe what we care for is the number of requests per unit of time that we can handle. Now in the context of that same application if we think about things from the client's perspective maybe its really the response time that can be used to evaluate whether something is better, or more useful than the alternative.

For these kinds of properties of the system, maybe I want to know their average value or whether their maximum or minimum values, so their best and worst case values, but also perhaps, I'm concerned with just, what is the number of requests per time that I can service or what is the response time that I can deliver to clients, and most of the time, 95% of the time or 99% of the time, so yes there might be a few outliers, a few situations in which my requests rate drops, but as long as 95% of the time its exactly where I want it to be, that's a solution that's good for me. So because of the fact that these outliers, these remaining 5%, may have very different behavior than the rest of the requests, or the rest of the time that the service is operating. Then when you're using the average numbers for these values, the evaluation may look very different than when we're using the 95 percentile values. Or maybe we're designing some hardware chip and, in that case, really from the hardware perspective the thing that we're really concerned with is whether or not the overall utilization of the hardware, the CPU, is better.

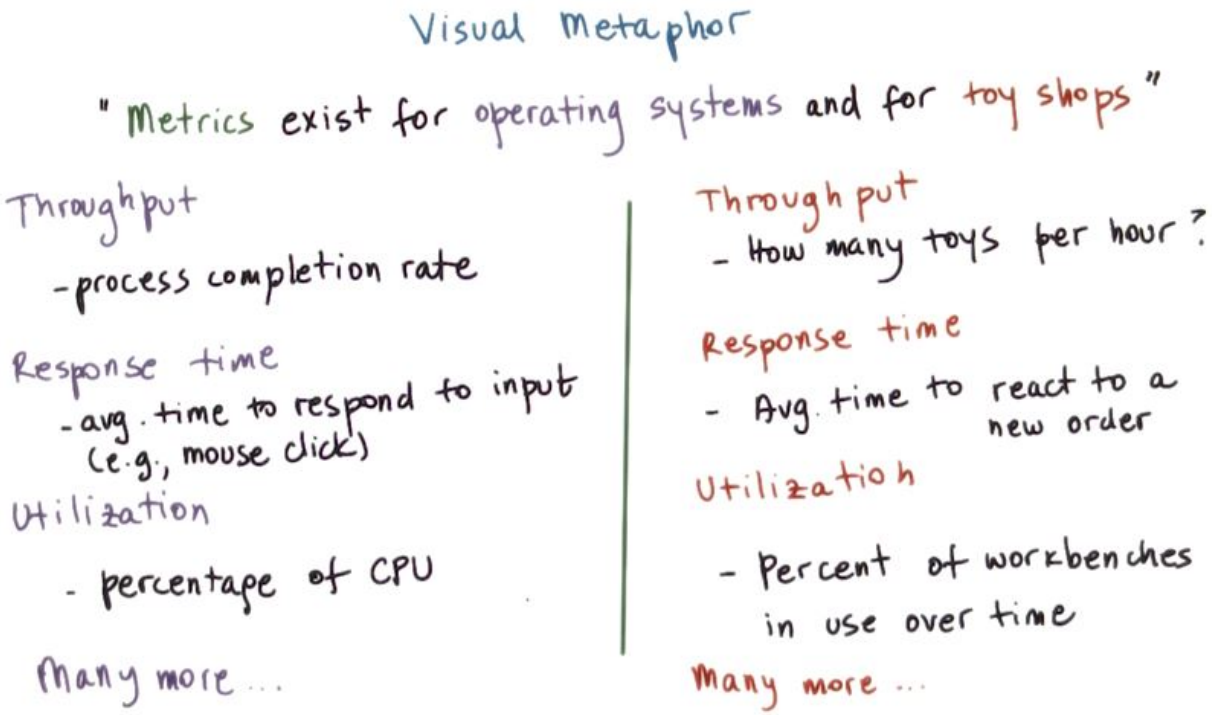


What these examples illustrate is that to evaluate some solution and to determine whether its useful or not, it is important to determine what are the properties that we really care for, what are the properties capture the utility of that particular solution. We call such properties ***metrics***, so basically the evaluation and the answer to whether something useful or not will depend on the relevant metrics.

04 - Visual Metaphor

Let's consider a visual metaphor in our discussion about metrics. We will do this by comparing metrics that exist in a toy shop, to metrics that exist in operating systems.

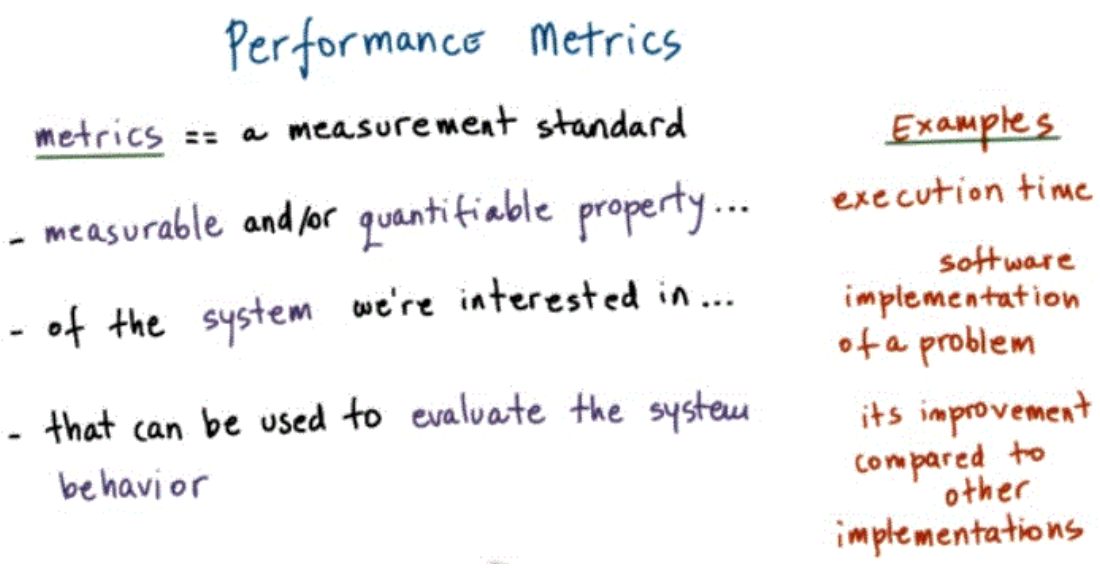
For instance, from the perspective of the toy shop manager, a number of properties of how the workers operate, how the toy shop is being run, may be relevant. One example is ***throughput***. The toy shop manager would want to make sure that this is as high as possible. Other things that may be important for the toy shop manager include how long does it take to react to a new order on average? Or what is the percentage of the workbenches that are used over a period of time? There can clearly be many more properties of the toy shop and how it's run that are relevant to the toy shop manager.



Metrics such as ***throughput***, ***response time***, ***utilization*** and others are also relevant from the operating systems perspective. For instance, it's important to understand how many processes can be completed over a period of time on a particular platform. It's important to know how responsive the system is. So when we click the mouse, does something happen immediately or we have to wait some noticeable amount of time? Does the operating system design lead to a solution that utilizes the CPU, devices, memory well, or does it leave a lot of unused resources? So metrics exist in any type of system, and it's important to have them well-defined when you're trying to analyze how a system behaves and how it compares to other solutions.

05 - Performance Metrics Intro

If you have not noticed yet performance considerations are really all about the metrics that we choose. Ideally, metrics should be represented with values that we can measure and quantify. The definition of the term ***metrics*** according to Webster's, for instance, is that its a measurement standard. In our analysis of systems a metric should be measurable, it should allow us to quantify a property of a system so that we can evaluate the system's behavior or at least compare it to other systems.



For instance, let's say we are concerned with the execution time of the system, thats a metric, we can measure it, we can quantify exactly what is the execution time of a system so it is a quantifiable property as well. A metric is allocated in some way with some system that we're interested in. For instance, that can be the implementation of a particular problem, the software implementation of a problem, and that's what we want to measure the execution time of. And a metric should tell us something about the behavior of the system we're interested in. For instance, it can tell us whether its an improvement over other implementations of the same problem. For the later, in order to perform this kind of evaluation and comparisons we really should explore the values of this metrics over some range of meaningful parameters by varying the workload that this implementation needs to handle or by varying the resources that are allocated to it or other dimensions.

06 - Performance Metrics

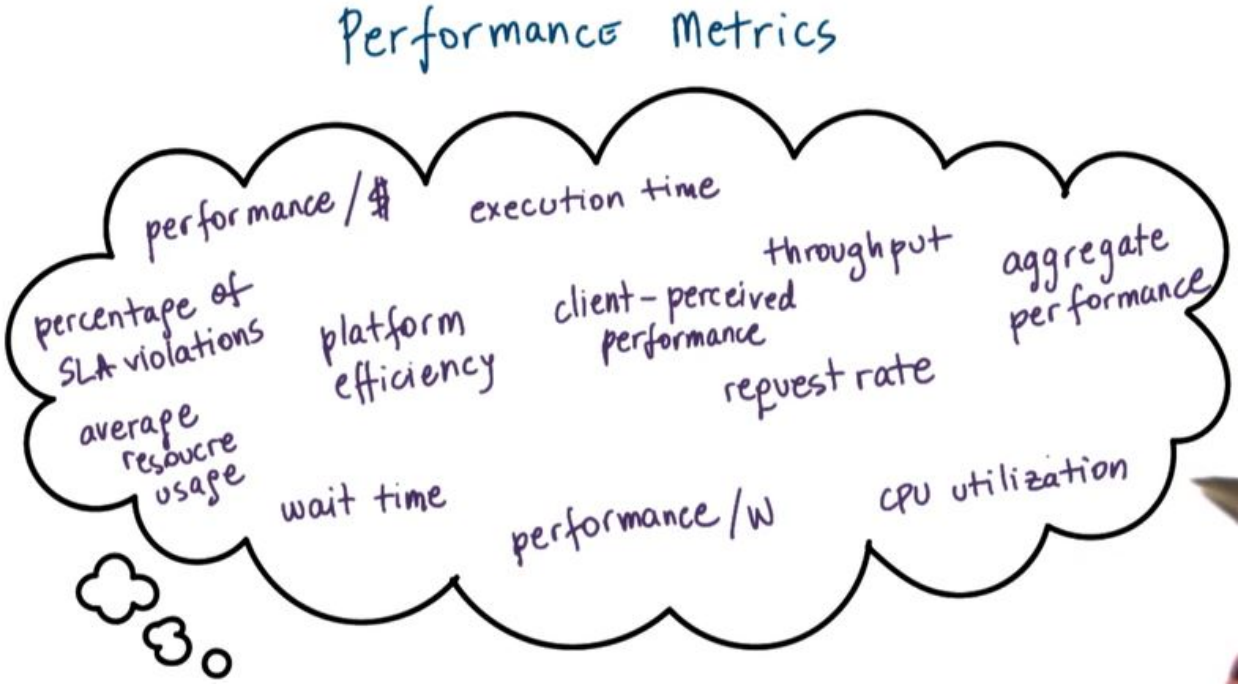
So far in the lesson we mentioned several useful metrics. For instance, we talked about execution time, throughput, response time, CPU utilization, but there are many other useful metrics to consider.

For instance, user may not just care when they will get an answer, but they may also care when their job will actually start being executed. We call this metric ***wait time***. If the job is interactive, so the user needs to interact with this, obviously the sooner it starts the sooner the user will be able to do something about it. If the job is a long-running job, then the sooner it starts the user has a chance to find out maybe somethings going wrong, so it can reconfigure the task, it can stop it and then reconfigure it and launch it again. So wait time could be an important metric in some contexts.

Then let's think about ***throughput***, for instance. We know throughput helps evaluate the utility of a platform, so how many tasks will it complete over a period of time, how many processes, how many job will be completed over a period of time. This can be relevant in the context of a

single machine, a single server, or in the context of an entire data center, for instance. Now if I'm the owner of the data center, throughput is not the only thing that I care for; I'm probably more concerned about some other type of metric that we can call ***platform efficiency*** and this says some combination of how well I utilize my resources to deliver this throughput, so its not just a matter of having higher throughput but also being able to utilize the resources that are available in my data center more efficiently. The reason for this is that as a data center operator I make money when I complete jobs, so the higher the throughput the greater the income for me. However, I also spend money to run the machines, to buy more servers, so its important to have a good ratio, so platform efficiently will capture that.

If its really just the dollars that I'm concerned about, then a metric like ***performance per dollar*** would capture that. So if I'm considering buying the next greatest hardware platform, then I can think about whether the cost that I will pay extra for that new piece of hardware will, basically, be compensated with some impact on the performance that I'm seeing. Or maybe I'm concerned about the amount of power (watts) that can be delivered to a particular platform or the energy that will be consumed during the execution, so then defining some metrics that capture ***performance per watt*** or performance per joule will be a useful ones.



You may have hear of the term ***SLA*** (***Service Level Agreement***), enterprise applications will give typically SLAs to their customers. One example, for instance, will be that you will get a response within 3 seconds. Or it may be even more subtle than that, for instance, service like Expedia, perhaps, has an SLA with its customers and its customers would be like Delta Airlines and American Airlines, that it will provide most accurate quote for 95% of the flights that are being returned to customers. So then for that enterprise application, one important thing would be whether there are any SLAs that are violated, whether there are any customer requests that

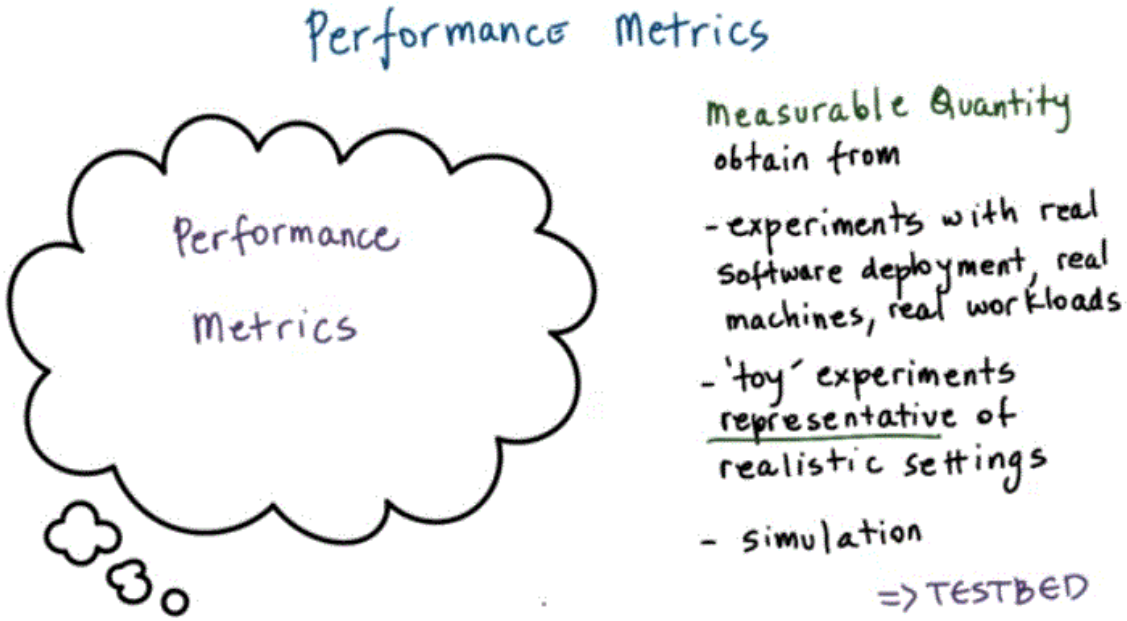
took longer than three seconds, or that did not provide quotes for airfare that were 100% accurate. A metric like percentage of SLA violations would capture that information.

For some applications, there is some natural opportunity for a slack in the application, for instance, if you think about a regular video application, humans cannot perceive more than 30 fps (frames per second) so being so focused on the fps and trying to maximize that fps rate, thats not the goal, however, making sure that there is at least 30 fps so that the users don't start seeing some random [buffering pauses] during the video they're watching on YouTube. Thats something thats important.

So its not so much about this raw request rate or wait time, but rather it is a metric that really is concerned whether the client perceives the service as performing well or not. You may be concerned with the performance metric of an individual application or you may be need to try to come up with some kind of ***aggregate performance*** metric that tries to average the execution time for all tasks or average the wait time for all tasks, or maybe even this would be a weighted average based on the priorities of the tasks. Also in addition to being just concerned with CPU utilization, there are a number of other resources that we may be concerned about, memory, file system, so the storage subsystem, so some metrics that are concerned with the ***average resource usage*** are also useful.

07 - Performance Metrics Summary

In summary, a metric is some measurable quantity that we can use to reason about the behavior of the system. Ideally, we will obtain these metrics, we will gather these measurements, running experiments using real software deployments on the real machines using real workloads. However, sometimes that really not an option, we cannot wait to actually deploy the software before we start measuring something about it or analyzing its behavior.

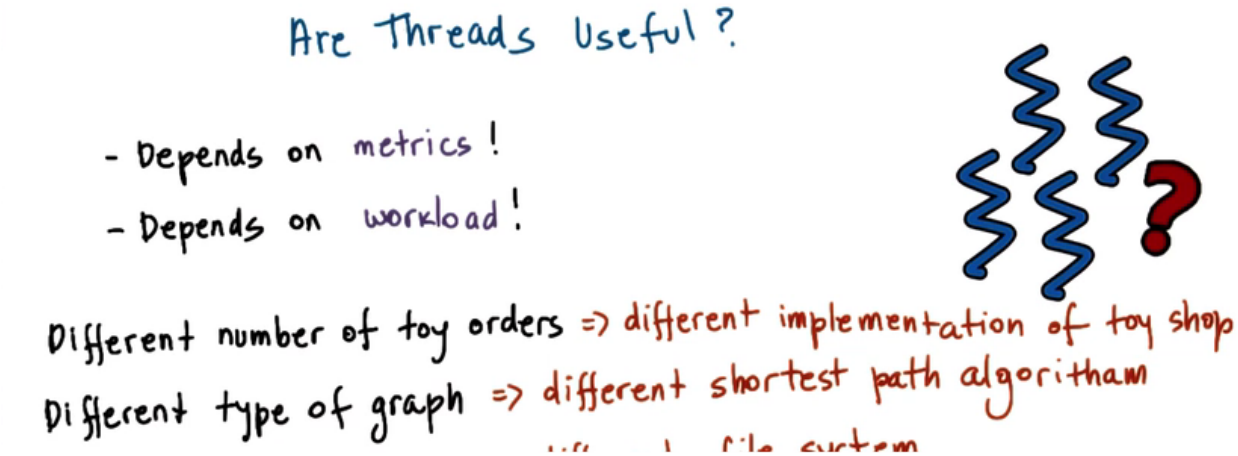


In those cases, we have to resort to experimentation with some representative configurations that in some way mimic, as much as possible, the aspects of the real system. The key here is that such a toy experiment must be representative of this real environment, so we must use workloads that have similar access patterns, similar types of machines, so as closely mimics the behavior of the real system as possible.

And possibly we will have to supplement those toys experiments with simulation so that we can perhaps create an environment that some how mimics up a larger system than what's possible with a small (toy) experiment. Any of these methods represent viable settings where one can evaluate a system and gather some performance metrics about it. We refer to these experimental settings as a ***testbed*** so the testbed that tells us where were the experiments carried out and what were the relevant metrics that were measured.

08 - Really Are Threads Useful

So if we go back now to our question, are threads useful? We realize that the answer is not so simple. We cannot simply say, yes, threads are useful. We know that the answer of the question will depend on the metrics that we're interested in. Also, it will depend on the workload. We saw in the toy shop example where we compared the boss worker and the pipeline model that the answer as to which model is better dependent on the number of toys that need to be processed, so the number of orders. So in the toy shop example, depending on the workload, the toy orders, and metrics we were concerned in, it lead us to conclusions that a different implementation of the toy shop, a different way to organize its workers was a better one.



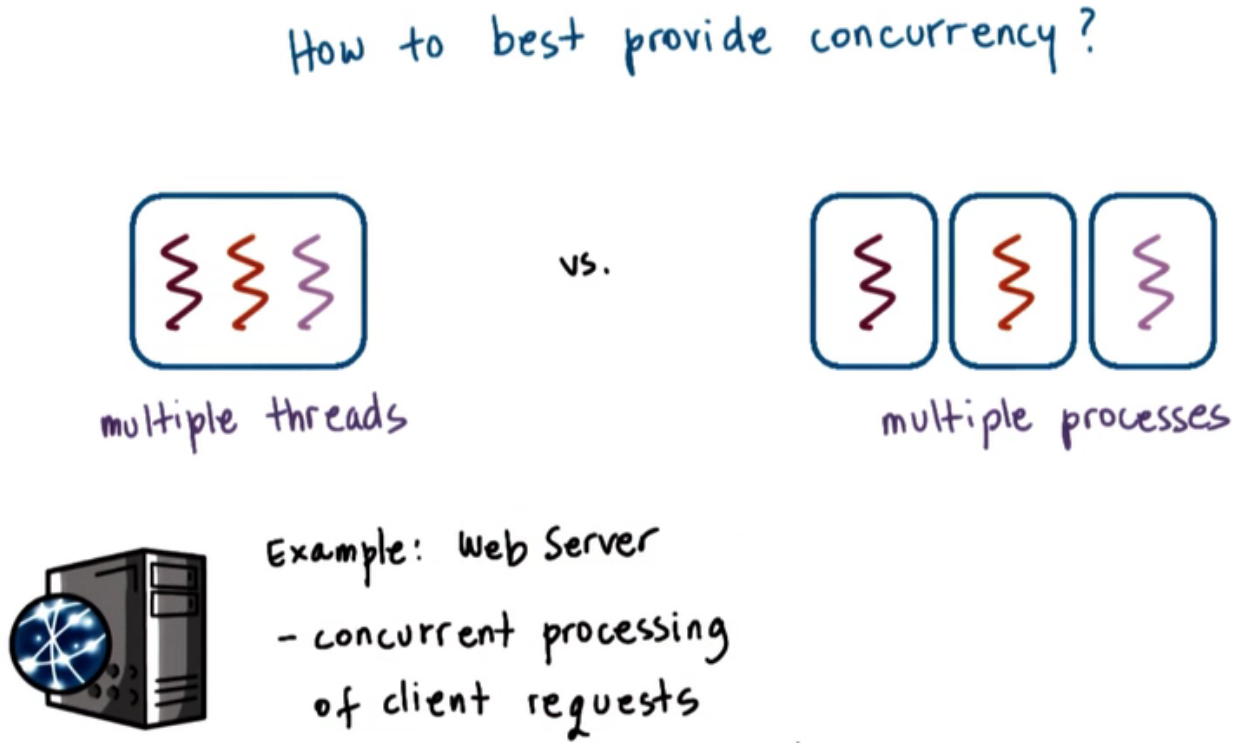
If you look at other domains, for instance, if we think about graphs and graph processing. Depending on the kind of graph, how well connected it is, it may be suitable to choose different type of shortest path algorithm. Some shortest path algorithms are known to work well on densely connected graphs whereas others work better for sparsely connected graphs. So again,

the workload is something that we're interested in. When comparing file systems, maybe what's important to consider is the, the patterns. The file, some file systems may be better for predominantly read accesses whereas others are better for more of a mixed workload, where files are both read and updated.

The point of looking at all of these is that across the board, both for the first question as well as in these other cases, the answer of whether something is better than an alternative implementation or an algorithm, it's pretty much always it depends. Depending on the file pattern, depending on the graph, depending on the number of toy orders.So similarly, the answer to, are threads useful isn't really going to be a straightforward yes and no one. It's really going to depend on the context in which we're trying to answer this question. And while we are at this, it depends answer, you should know that it's pretty much always the correct answer to a question in systems. However, it's never going to be an accepted one. I will **not** take it as accepted answer in this course either.

For the remainder of this lecture, we will to answer a specifically, whether threads are useful. And when are threads more or less useful when comparing a multithreaded-based implementation of a problem to some alternatives. I will also provide you with some guidance on how to define some useful metrics, and how to structure experimental evaluations, so that you can correctly measure such metrics.

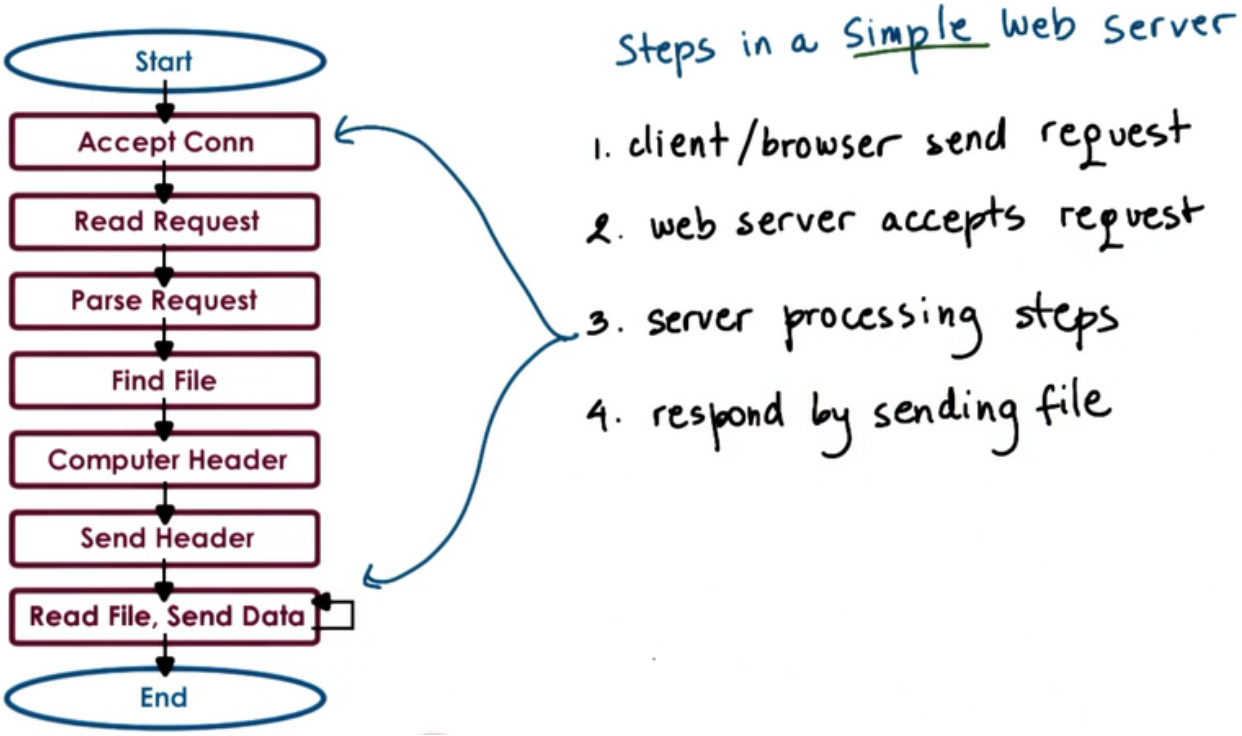
09 - Multi Process vs Multi Threaded



To understand when are threads useful, let's start to think about what are the different ways to provide concurrency and what are the trade-offs among those implementations. So far we've

talked about multi-threaded applications, but an application can be implemented by having multiple concurrently running processes. We mentioned this in the earlier lecture on Threads and Concurrency. So let's start by comparing these two models. To make the discussion concrete, we'll do this analysis in the context of a web server. And for web server its important to be able to concurrently process client requests, so that is the concurrency that we care for there.

Before we continue, let's talk for a second about what are the steps involved in the operation of a simple web server. At the very first, the client or browser needs to send a request that the web server will accept. So let's say that this is a request to www.gatech.edu and the web server at Georgia Tech needs to accept that request.



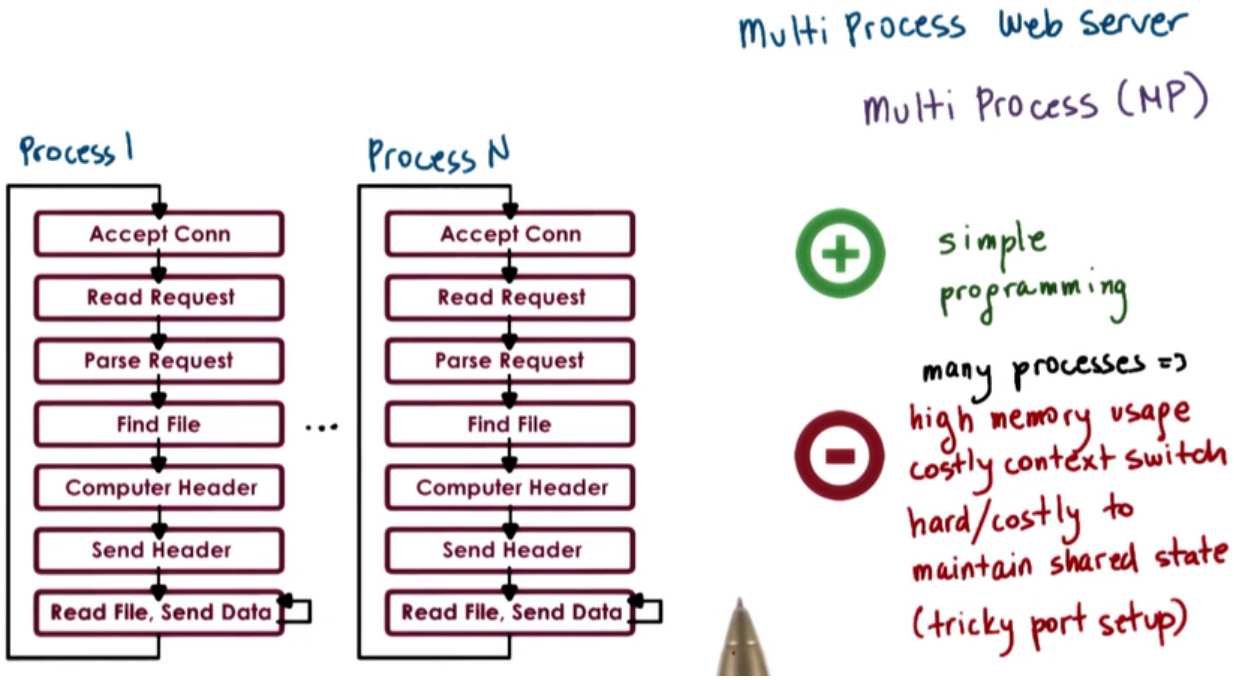
After the request is accepted, there are a number of processing steps that the web server needs to perform before finally responding with the file. Now we will talk about a simple web server, so if we take a look at what these steps are. So we accept a connection, we read the request (so there is an http request that's received), and we need to parse that request. We need to then find the file in the local file system, that's at the server side. Once we extract the file, we need to compute the header, send out the header, and then also send out the file, or potentially send out an error message along with the header that the file is not found.

So for the rest of this lesson, we'll really focus on this simple web server processing. One of the things that's worth pointing out is that there's some differences among these steps. Some of them are more computationally intensive, so its mostly the work is done by the CPU. For instance, parsing the request or computing the header, this is mostly done by the CPU. Other steps may require some interaction with the network, like accepting connections, reading

request, or sending the data. Or the disk, for instance, when finding the file and reading the file from disk. These steps may potentially block, but whether they block will really depend on what is the state of the system at a particular point of time. So for instance, the connection may already be pending or the data for the file may already be cached in memory because of the previous request that serviced that file. So in those cases, these will not result in an actual call to the device, so an actual implication of the disk or the network and will be serviced much more quickly. Once the file, or potentially the error message, are sent out to the client then the processing is complete

10 - Multi Process Web Server

This (Process 1 below) then clearly represents a single threaded process. One easy way to achieve concurrency is to have multiple instances of the same process and that way we have a ***multi-process (MP)*** implementation. This illustration is adapted from Vivek Pai's paper "Flash: An Efficient and Portable Web Server" and it appears as Figure 2 in the paper.



The benefits of this approach is that it is simple, once we have correctly developed this sequence of steps for one process, we just spawn multiple processes. There is some down sides, however, with running multiple processes on a platform. We'll have to allocate memory for every one of them and this will ultimately put high load on the memory subsystem and will hurt performance.

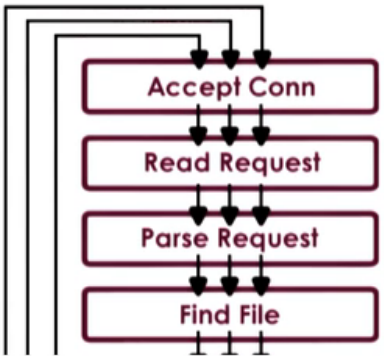
Given that these are processes, we already talked about the cost of context switching among processes. Also it can be rather expensive to maintain shared state across processes because the communication mechanisms and the synchronization mechanisms that are available across processes; those are a little bit higher overhead. And in some cases it may even be a little bit

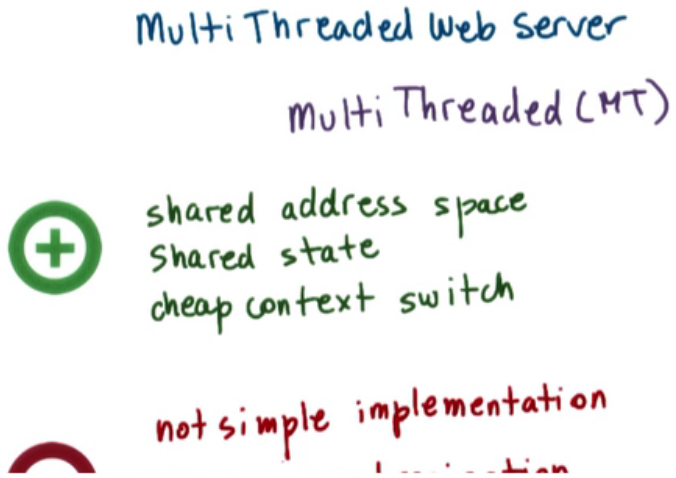
tricky to do certain things like, for instance, forcing multiple processes to be able to respond to a single address and to share an actual socket port.

11 - Multi Threaded Web Server

An alternative to the multi-process model is to develop the web server as a ***multi-threaded (MT)*** application. So here (pointing at looping lines on the left side of the illustration below) we have multiple execution contexts, multiple threads within the same address space, and every single one of them is processing a request. Again this illustration is taken from Pai's Flash paper and this is Figure 3 there. In this figure, every single one of the threads executes all the steps, starting from the accept connection call all the way down to actually sending the file.

Another possibility is to have the web server implemented as a Boss-Workers model where a single boss thread performs the accept connection operation and every single one of the workers performs the remaining operations, from the reading of the http request that comes in on that connection, until actually sending the file.



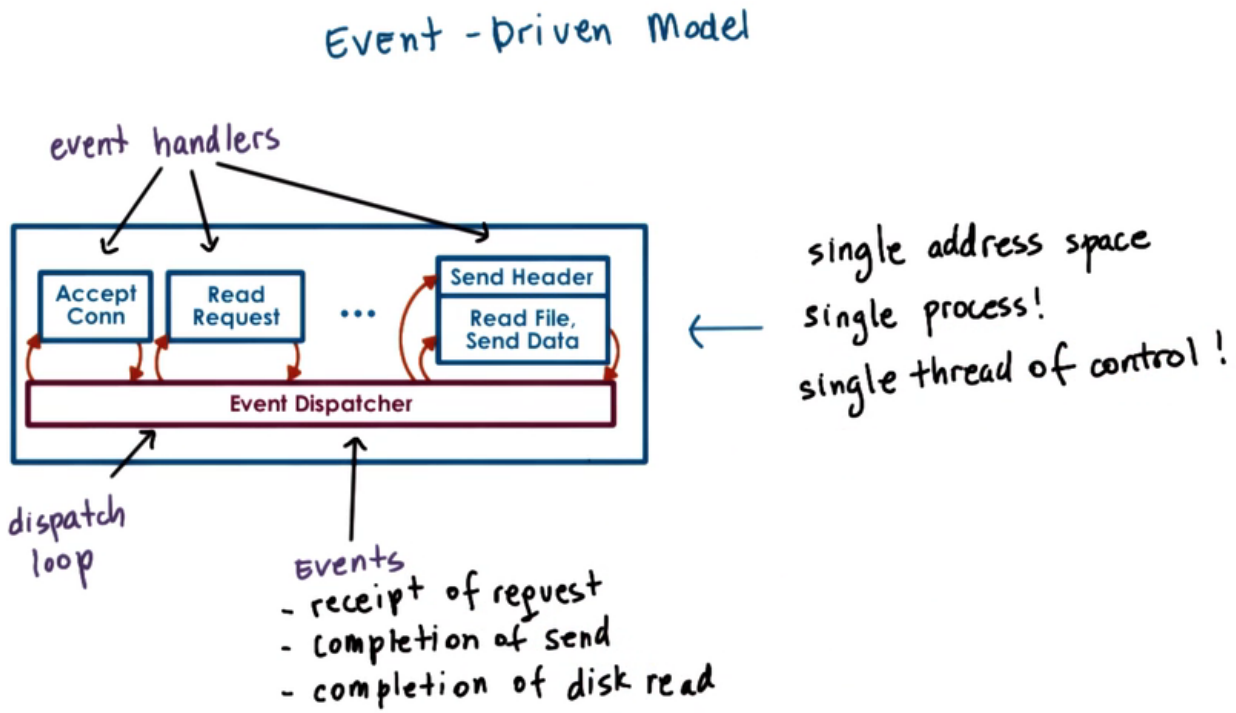


The benefits of this approach is that, thread share the address space, so they will share everything that's within it. They don't have to perform system calls in order to coordinate with other threads like what's the case in the multi-threaded (errata: MP) execution. Also context switching between these threads is cheap, it can be done at the multi-threading library level. Because a lot of the per thread state is shared among them, then we don't have to allocate memory for everything that's required for each of these execution contexts, they share the same address space, so the memory requirements are also lower for the MT application.

The downside of the approach is that it is not so simple and straightforward to implement the MT program; you have to explicitly deal with synchronization when threads are accessing and updating shared state. And we also rely for the underlying operating system to have support for threads; this is not so much of issue today, operating systems are regularly multi-threaded, but it was at the time of the writing of the Flash paper, so we will make sure to address this argument as well in our explanations.

12 - Event-Driven Model

Now let's talk about an alternative model for structuring server applications that perform concurrent processing. The model we'll talk about is called ***event-driven model***. An event-driven application can be characterized as follows. The application is implemented in a single address space, there is basically only a single process and a single thread of control. Here is the illustration of this model and this is taken from Vivek Pai's Flash paper as well.

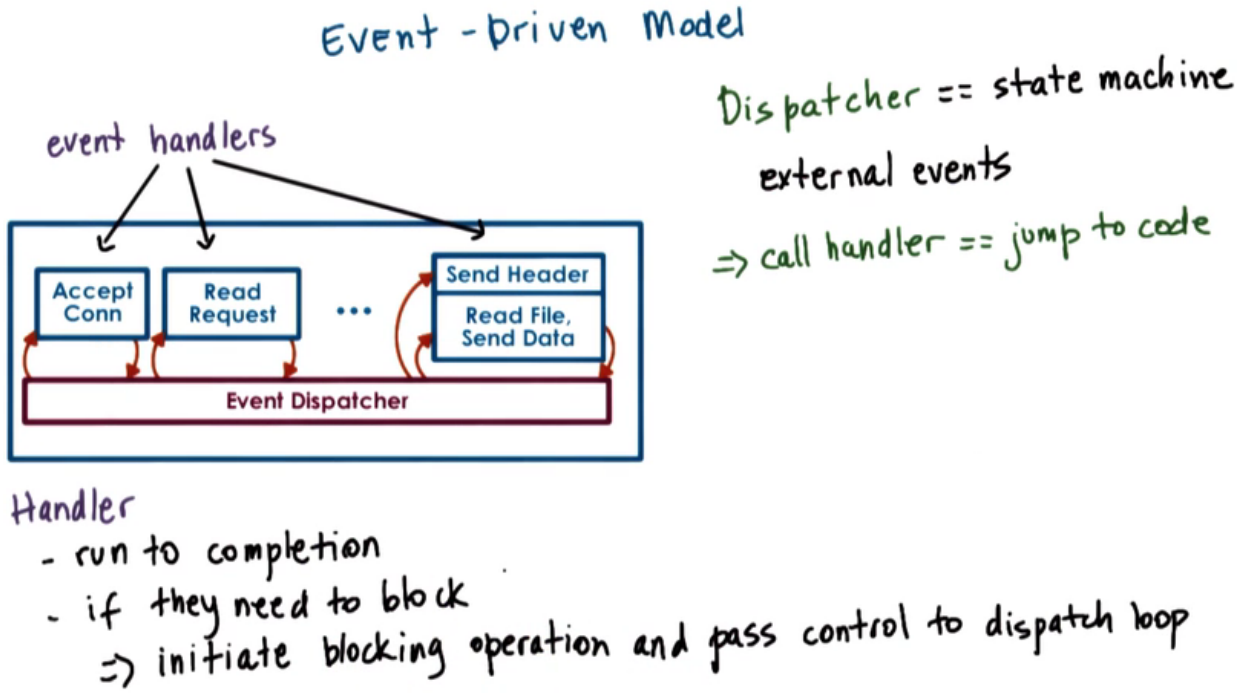


The main part of the process is an event dispatcher that continuously, in a loop, looks for incoming events and then based on those events invokes one or more of the registered [event] handlers. Here events correspond to some of the following things. Receipt of a request from the client browsers, that message thats received from the network, thats an event. Completion of send, so once the server responds to the client request, the fact that the send completed, thats another event as far as the system is concerned. Completion of a disk read operation, thats another event that the system will need to know how to handle.

The dispatcher has the ability to accept any of these types of notifications, and then based on the notification type, to invoke the appropriate handler. So in that sense it operates very much like a state machine. Since we're talking about a single-threaded process, invoking a handler

simply means that we will jump to the appropriate location in the processes address space where the handler is implemented. At that point, the handler execution can start.

For instance, if the process is notified that there is a pending connection request on the network port that it uses, the dispatcher will pass that event to the accept-connection handler. If the event is a data of messages on an already established connection, then the event dispatcher will pass that to the read-request handler. Once the filename is extracted from the request and its confirmed that the file is present, the process will send out chunks of the file and then once there is a confirmation that that chunk of the file, portion of the file, has been successfully sent, then it will continue iterating over the handler thats dealing with the send operation. If the file is not there then some sort of error message will sent to the client.



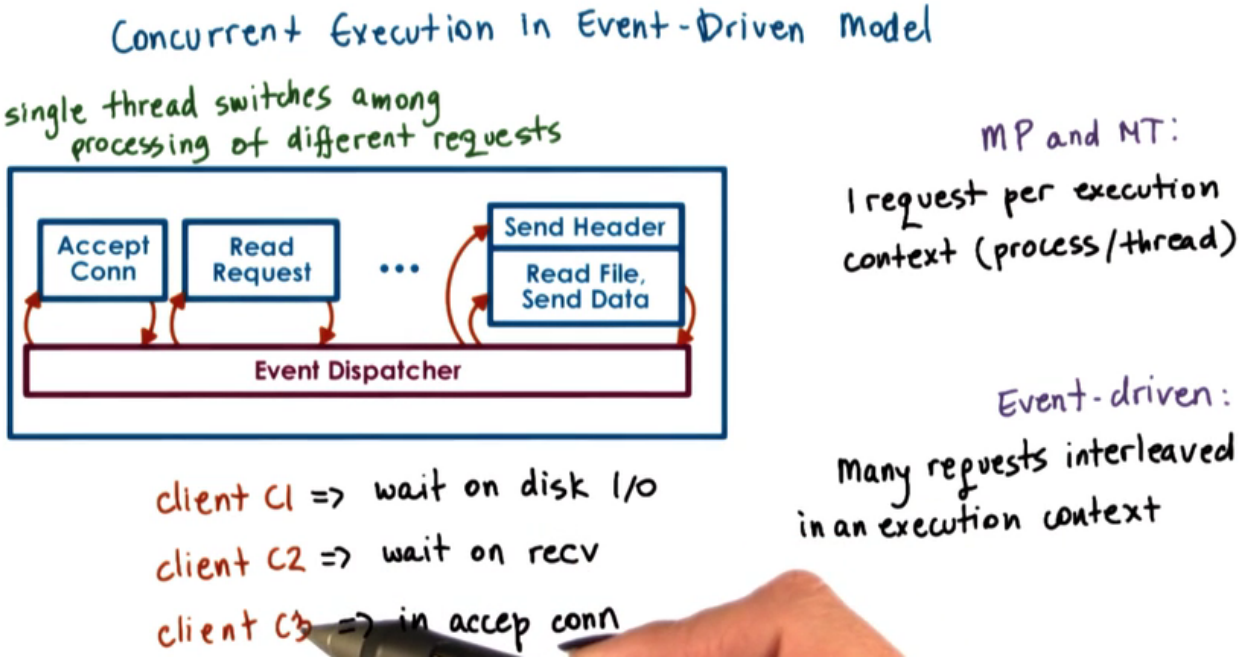
So whenever an event occurs, the handlers are the sequence of code that executes in response of these events. The key feature of the handlers is that they run to completion. If a handler needs to perform a blocking operation, it will initiate the blocking operation and then it will immediately pass control back to the event dispatcher, so we'll no longer be in the handler. At that point, the dispatcher is free to service other events or call other handlers.

13 - Concurrency in the Event Driven Model

You're probably asking yourselves, if the event-driven model has just one thread then how do we achieve concurrency? In the multi-process (MP) and the multi-threaded (MT) models, we had each execution context, whether its a process or a thread, handle only one request at a time. To achieve concurrency, we would simply add multiple execution contexts, so multiple processes or multiple threads, and then if necessary, if we have fewer CPUs than context, then we would have to context switch among them.

The way the event-driven model achieves concurrency is by ***interleaving*** the processing of multiple requests within the same execution context. Here in the event-driven model, we have a single thread and the single thread switches its execution among the processing thats required for different requests. Let's say we have a client request coming into the system, so its a request for client C1, and we receive a request for a connection that gets dispatched, the accept operation gets processed. Then we receive the actual request, so the http message, that gets processed, the message gets parsed, we extract the files, so now we actually need to read the file, and we initiate I/O from the reading-file handler. So at that point, the request for client C1 has been processed through several of these steps and its waiting on the disk I/O to complete.

Let's in the meantime, two more requests have come in, so client C2 and C3 have sent a request for a connection. Let's say the C2 request was picked up first, the connection was accepted, and now for the processing of C2, we need to wait for the actual http message to received. So the processing of C2 is waiting on an event from the network that will have the http message that needs to be received. And let's say C3, its request has been accepted and its currently being handler, so the C3 request is in the accept-connection handler.



Some amount of time later the processing of all of these three requests has moved a little bit further along, so the request for C3 was completed, and now that request is waiting on an event with the http message. The request for C2, that one perhaps we're waiting on the disk I/O in order to read the file that needs to be sent out. And maybe the request for C1 already started sending the file in chunks at a time, so blocks of some number of bytes at a time, so its waiting in one of those iterations.

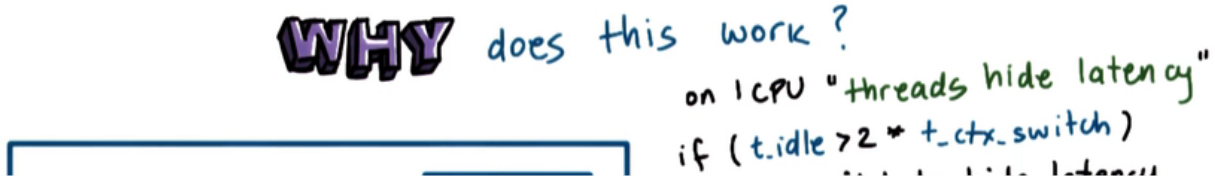
So although we have only one execution context, only one thread, if we take a look, we have concurrent execution of multiple client requests. It just
happens to be interleaved, given that there's one

execution context, however, there are multiple, at the same time multiple client requests being handled.

14 - Event-Driven Model Why

The immediate question is why does this work? What is the benefit of having a single thread thats just going to be switching among the processing of different requests compared to simply assigning different requests to different execution contexts, to different threads or even to different processes.

Recall our introductory lecture about threads in which we said that on a single CPU, threads can be useful because they help hide latency. The main take-away from that discussion was that, if a thread is going to wait more than twice the amount of time it takes to perform a context switch (t_idle > 2 * t_ctx_switch), then it makes sense to go ahead and context switch to another thread that will do some useful work. And in that way, we hide this waiting latency. If there really isn't any idle time (t_idle == 0), so if the processing of a request doesn't result in some type of blocking I/O operation on which it has to wait, then there are no idle periods, it doesn't make sense to context switch. The context switching time will be just cycles that are spent on copying and restoring of threads, or of process information, and those cycles could have been much better spent actually performing request processing.



So in the event-driven model, a request will be processed in the context of a single thread as long as it doesn't have to wait. Whenever a wait needs to happen then the execution thread will switch to servicing another request. If we have multiple CPUs, the event-driven model still makes sense, especially when we need to handle more concurrent requests than the number of CPUs. For instance, each CPU could host a single event-driven process, and then handle multiple concurrent requests within that one context. And this could be done with less overhead than if each of the CPUs had to context switch among multiple processes or multiple threads, where each of those is handling a separate request.

There is one gotcha though here, it is important to have mechanisms that will steer, that will direct, the right set of events to the appropriate CPU, to the appropriate instance of the event driven process. And there are mechanisms to do this and there's current support in networking hardware to do these sorts of things, but I'm not going to go into this in any further detail. So just know that overall as a model, this is how the event-driven model would be applied to a multi-CPU environment.

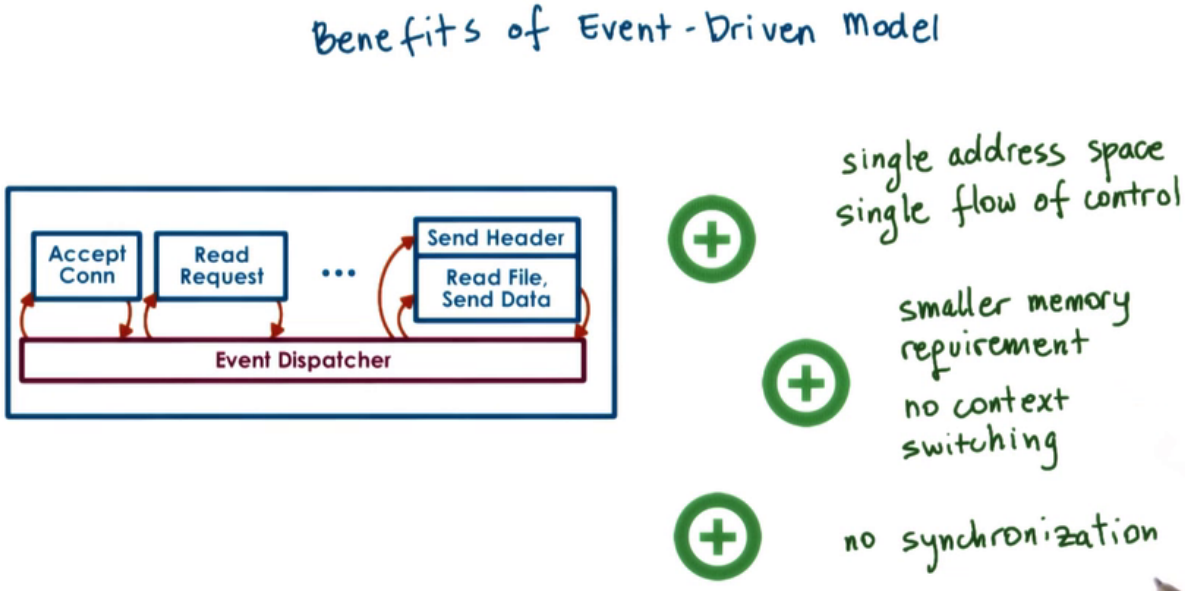
15 - Event-Driven Model How

Now let's see how can this be implemented? So at the lowest level, we need to be receiving some events, some messages from the network or from the disk, so information about completed requests to read a portion of the file or write the file, etc. The operating systems use these two abstractions (sockets/network, files/disk) to typically represent networks or disks, so sockets are typically used to represent an interface to the network, and then files are what's really stored on disk, so these are the main abstractions when it comes to storage.

Now although they're called differently, sockets and files, it is quite fortunate that internally the actual data structure that's used to represent these two different abstractions, is actually identical. Its called a ***file descriptor*** (FD). So then an event in the context of this web server, is an input on any of the FDs that are associated with it, so on any of the sockets or any of the files that are being accessed by the connections that these sockets carry.

To determine which FD has input, so to determine that there is an event that has arrived in this system, the Flash paper talks about using the select() call. The select() call takes a range of FDs, and then returns the very first one that has some kind of input on it. And that is regardless of whether that FD is a socket or a file ultimately. Another alternative to this is to use a poll() API, so this is another system call thats provided by current operating systems. The problem with both of these is that they really have to scan through potentially really large lists of FDs until

they find one. And it is very likely that among that long list of FDs, they're going to be only very few that have input, so a lot of that search time will be wasted. An alternative to these is a more recent type of API that supported by, for instance, the Linux kernel, and thats epoll(), so this eliminates some of the problems that select() and poll() have and a lot of the high performance servers that require high data rates and low latency use this kind of mechanism today.

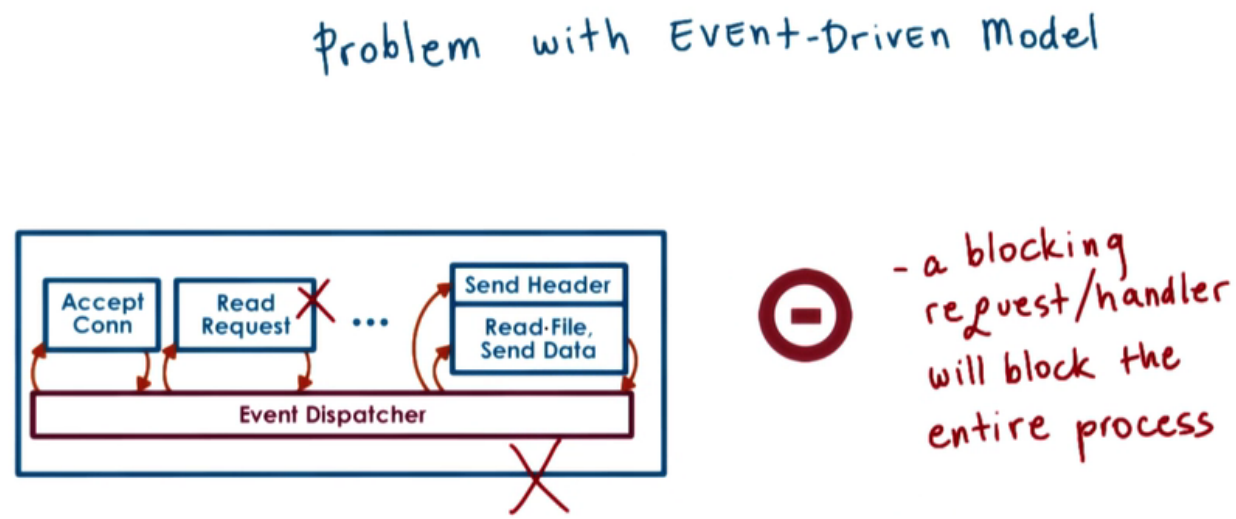


The benefits of the event-driven model really come from its design, its a single address space, single flow of control. As a result, the overheads are lower, there is no need for context switching. Overall its a much more compact process, so it has smaller memory requirements and the programming is simpler, we don't need to worry about use of synchronization primitives, about shared access to variables, etc.

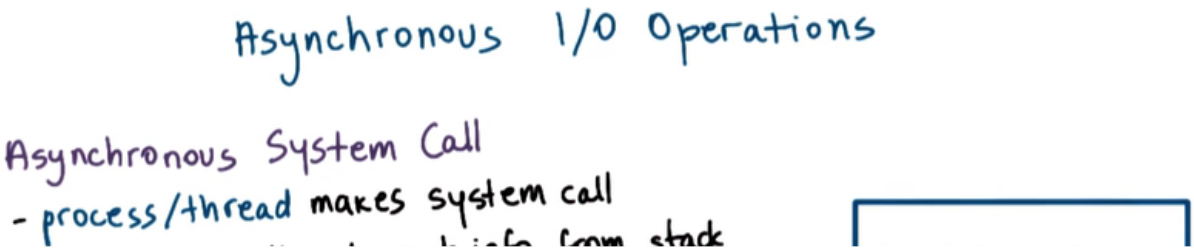
Now in the context of this single thread, we are switching among multiple connections, so we are jumping all over the code-base of this process and executing different handlers, accessing different state. That will have some effects on, basically, loss of localities and cache pollution effects, however, that will be significantly lower than what would have been happening if we were doing a full-blown context switching. So the overheads and some of the elimination of the synchronization, these are some of the things that really make this an attractive approach.

16 - Helper Threads and Processes

The event-driven model doesn't come without any challenges. Recall that when we talked about the many-to-one multi-threading model, we said that a single blocking I/O call thats coming from one of the user-level (UL) threads, can block the entire process although there may be other UL threads that are ready to execute. A similar problem can occur here as well. If one of the handlers issues a blocking I/O call to read data from the network or from disk, the entire event-driven process can be blocked.

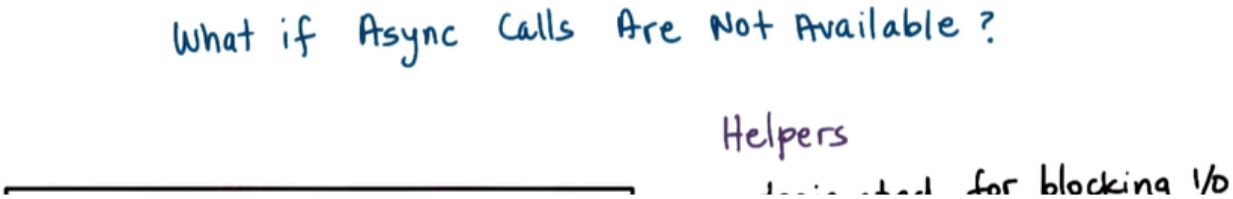


One way to circumvent this problem is to use asynchronous I/O operations. Asynchronous calls have the property that when the system call is made, the kernel captures enough information about the caller and where and how the data should be returned once it becomes available. Async calls also provide the caller with an opportunity to proceed executing something and then come back at a later time to check if the results of the asynchronous operation are already available. For instance, the processor to thread can come back later to check if a file has already been read and the data is available in a buffer in memory.



One thing that makes asynchronous calls possible, is that the OS kernel is multi-threaded, so while the caller thread continues execution another kernel thread does all the necessary work and all the waiting thats needed to perform the I/O operation, to get the I/O data, and then to also make sure that the results become available to the appropriate UL context. Also asynchronous operations can benefit by the actual I/O devices. For instance, the caller thread

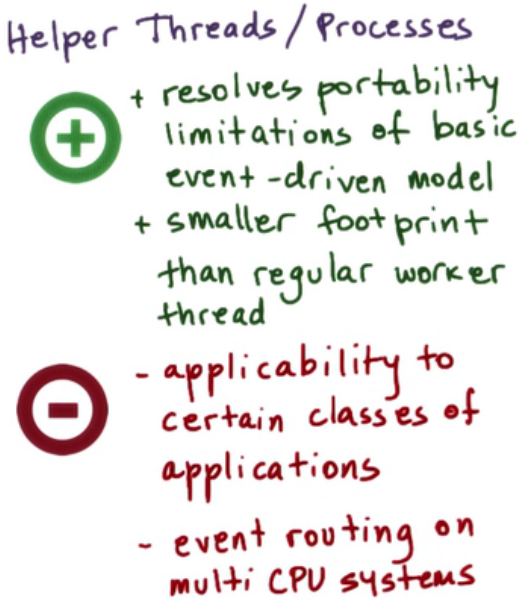
can simply pass some request data structure to the device itself, and then the device performs the operation and the thread, at a later time, can come and check to see whether the device has completed the operation. We will return to asynchronous I/O operations in a later lecture, what you need to know for now is that, when we're using asynchronous I/O operations our process will not be blocked in the kernel when performing I/O. In the event-driven model if the handler initiates an asynchronous I/O operation for network or for disk, the operating system can simply use the mechanism like select() or poll() or epoll() that we mentioned before, to catch such events. So in summary, asynchronous I/O operations fit very nicely with the event-driven model.



The problem with asynchronous I/O calls is that they weren't ubiquitously available in the past, and even today they may not be available for all types of devices. In a general case, maybe the processing that needs to be performed by our server isn't to read data from a file where there are asynchronous system calls, but instead maybe to call processing on some excellerator, some device that only the server has access to.

To deal with this problem, Vivek Pai's paper proposed the use of ***helpers***. When a handler needs to issue an I/O operation that can block, it passes it to the helper and returns to the event dispatcher. The helper will be the one that will handle the blocking I/O operation and interact with the dispatcher as necessary. The communication with the helper can be via socket-based interface or via another type of messaging interface thats available in operating systems called ***pipe***. And both of these present a file descriptor like interface, so the same kind of select() or poll() mechanism that we mentioned can be used for the event dispatcher to keep track of various events that are occurring in the system. This interface can be used to track whether the helpers are providing any kind of events to the event dispatcher. In doing this, the synchronous I/O call is handled by the helper, the helper will be the one that will block, and the main event dispatcher and the main process will continue uninterrupted. So this way, although we don't

have asynchronous I/O calls, through the use of helpers we achieve the same kind of behavior as if we had asynchronous calls. At the time of the writing of the paper, another limitation was that not all kernels where multi-threaded, so basically not all kernels supported the one-to-one model that we talked about. In order to deal with this limitation, the decision in the paper was to make these helper entities processes. Therefore, they call this model ***AMPED*** (***Asymmetric Multi-Process Event-Driven Model***), its an event-driven model, it has multiple processes and these processes are asymmetric, the helper ones only deal with blocking I/O operations, and then the main one performs everything else. In principle, this same kind of idea could have applied in a multi-threaded scenario where the helpers are threads not processes, so ***AMTED*** (***Asymmetric Multi-Threaded Event-Driven Model***), and in fact there is a follow-on on the Flash work that actually does this exact thing, the AMTED model.

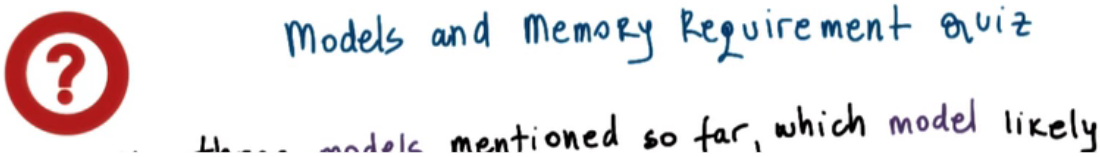


The key benefits of the asymmetric model that we described, is that it resolves some of the limitations of the pure event-driven model in terms of what is required from the operating system, the dependence on asynchronous I/O calls and threading support. In addition, this model lets us achieve concurrency with a smaller memory footprint than either the MP or MTing model. In the MP/MT model a worker has to perform every thing for a full request, so its memory requirements will be much more significant than the memory requirements of a helper entity. In addition, with the AMPED model we will have a helper entity only for the number of concurrent blocking I/O operations whereas in the MT or MP models we will have as many concurrent entities, as many processes or as many threads, as there are actual concurrent requests regardless of whether they block or not.

The downside is that although this works well with the server type applications, it is not necessarily as generally applicable to arbitrary applications. In addition, there are also some complexities with the routing of events in multi-CPU systems.

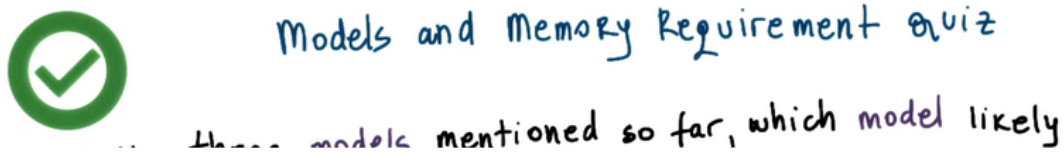
17 - Models and Memory Quiz

Here is a quick quiz analyzing the memory requirements of the three concurrency models we talked about so far. The question is of the three models mentioned so far, which model likely requires the least amount of memory? The choices are the Boss-Worker model, the Pipeline model, and the event-driven model. Also answer why you think that this model requires the least amount of memory, to see if your reasoning matches ours.



18 - Models and Memory Quiz

The correct answer is that likely the event-driven model will consume the least resources. Recall that in the other models we had a separate thread for each of the requests or for each of the pipeline stages. In the event-driven model we have handlers which are just procedures in that address space and the helper threads only occur for blocking I/O operations.

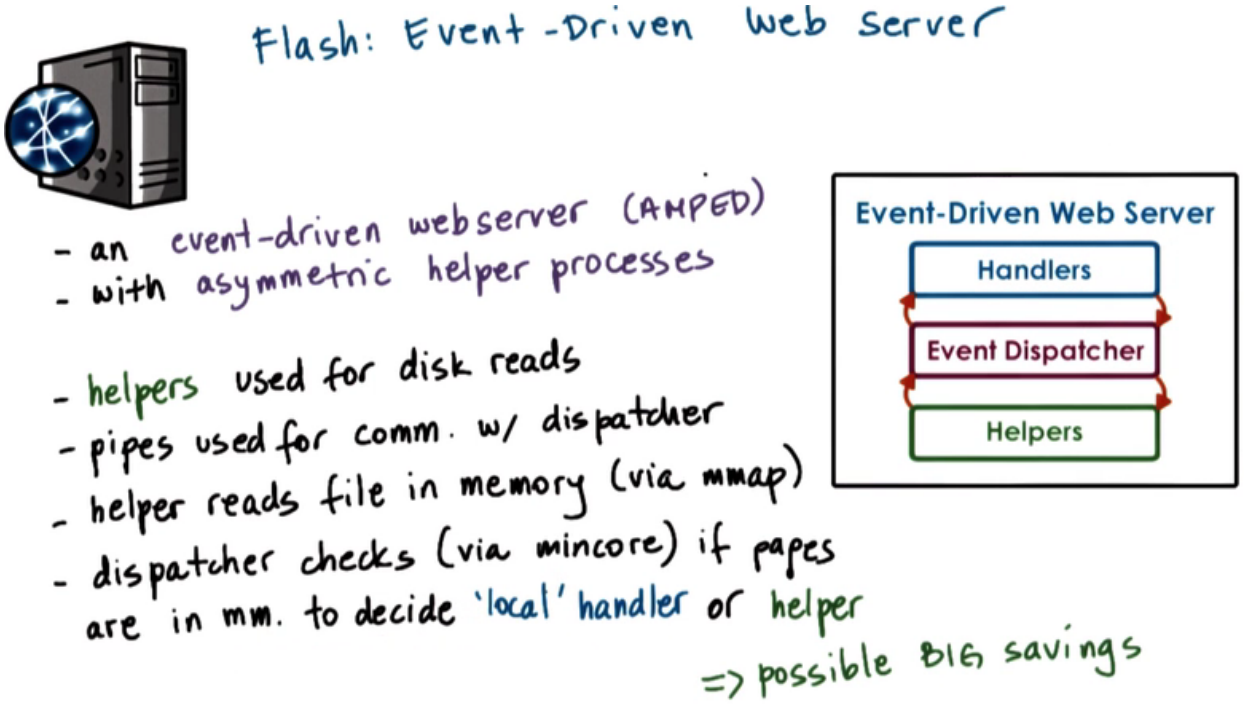


For the event-driven model, extra memory (mm?) is required only for the helper threads that are associated with concurrent blocking I/O calls. In the Boss-Worker model, extra memory will required for threads for all concurrent requests. And similarly even in the Pipeline model,

concurrent requests will demand multiple threads to be available in a stage of the pipeline if the level of concurrency is beyond the number of pipeline stages. As a result, the event-driven model will likely have the smallest memory footprint.

19 - Flash Web Server

With all of this background on the event-driven model, we will now talk about the Flash paper. Flash is an event-driven web server that follows the AMPED model, so basically it has asymmetric helper processes to deal with the blocking I/O operations. In the discussions so far we really described the architecture of Flash, so it uses helper processes for blocking I/O operations and then everything else implemented as an event dispatcher with handlers performing different portions of the web servicing tasks. Given that we're talking about a web server, and this is the old fashioned web 1.0 technology where basically the web server just returns static files, the blocking I/O operations that are happening in the system are really just disk reads, so the server just reads files that the clients request.



The communication from the helpers to the event dispatcher is performed via pipes. The helper reads the file in memory via the mmap call, and then the dispatcher checks, via an operation mincore, if the pages of the file are in memory, and it then uses this information to decide if it should just call one of the local handlers or if it should pass the request to the helper. As long as the file is in memory, reading it, it won't result in a blocking I/O operation and so passing it to the local handlers is perfectly okay. Although this is an extra check that has to be performed before we read any file, it actually results in big savings because it prevents the full process from being blocked if it turns out that a blocking I/O operation is necessary.

Flash: Additional Optimizations

Now we will outline some additional detail regarding some of the optimization that Flash applies, and this help us later understand some of the performance comparisons. The important thing is that these optimizations are really relevant to any web server. First of all, Flash performs application-level caching at multiple levels, and it does this on both data and computation. What we mean by this is, it common to cache files, this is what we call ***data caching***. However, in some cases it makes sense to cache computation, so in the case of the web server the requests are requests for some files. These files need to be repeatedly looked up so we need to find the file, traverse the directory, look up some of the directory data structures, that processing will compute some result, so some location, some pathname for the file and we will just cache that. We don't have to recompute that and look up the same information next time a request for that same file comes in.

Similarly in the context of web processing, the http header that files have, that are returned to the browser, its really going to depend on the file itself, so a lot of the fields in there are file dependant given that the file doesn't change, the header doesn't have to change so this is another type of application-level caching that we can perform and Flash does this. Also Flash does some optimizations that take advantage of the networking hardware, and of the network interface card. For instance, all of the data structures are aligned so that its easy to perform DMA operations without copying data. Similarly, they use DMA operations that have scatter-gather support, and that really means that the header and the actual data don't have to be aligned, one next to the other in memory, they can be sent from different memory locations, so there is a copy thats avoided. All of these are useful techniques and are now fairly common optimizations, however, at the time the paper was written, they were pretty novel and in fact some of the system they compared against did not have some of these things included.

20 - Apache Web Server

Before we continue, I would like to briefly describe the Apache web server, its a popular open-source web server, and its one of the technologies that, in the Flash paper the authors compare against. My intent is not to give a detailed lecture on Apache, thats beyond the scope of the course, but instead I want to give you enough about the architecture of Apache and how it compares to the models that we discussed in the class. And also the other way around, to understand how these discussions in class are reflected in real world designs.

Image Errata: skelethon = skeleton

From a very high-level the software architecture of Apache looks like this (box above). The core component provides the basic server-like capabilities, so this is accepting connections and managing concurrency. The various modules correspond to different types of functionality that is executed on each request. Specific Apache deployment can be configured to include different types of modules. For instance, it can have certain security features, some management of dynamic content, or even some of the modules are really responsible for more basic http request processing.

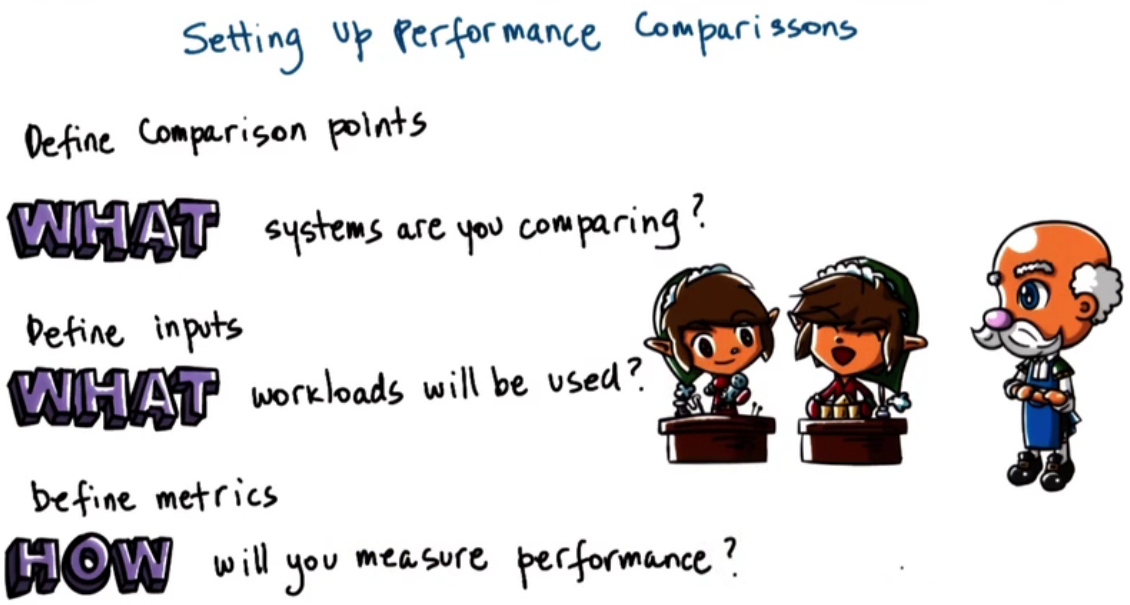
The flow of control is sort of similar to the event-driven model that we saw in the sense that each request passes through all of the modules. Like in the event-driven model each request ultimately passes through all the handlers. However, Apache is a combination of a MP and a MT model. In Apache, a single process, a single instance, is internally a MTed Boss-Workers process that has dynamic management of the number of threads. There's some configurable thresholds that can be used to dynamically track when to increase or decrease the number of threads in the pool. The total number of processes, so the MP part of the model, can also be dynamically adjusted and for these its information such as number of outstanding connections,

number of pending requests, CPU usage, a number of factors can drive how the number of threads per process and the total number of processes are adjusted.

21 - Experimental Methodology

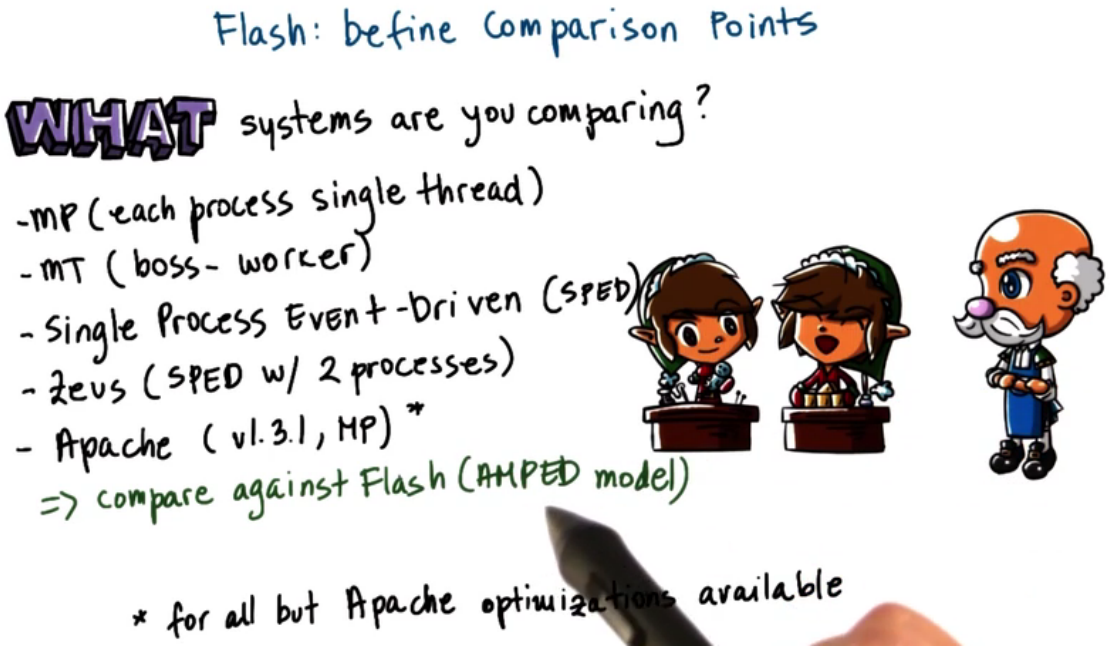
It is now time to discuss the experimental approach in the Flash paper. In the paper, the experiments are designed so that they can make stronger arguments about the contributions that the authors claim about flash. And this is something that you should always consider when designing experiments, that they should help you with the arguments that you're trying to make. To do this, to achieve a good experimental design, you need answer a number questions.

For instance, you should ask yourself what is it that you're actually comparing? Are you comparing two software implementations? Keep the hardware the same. Are you comparing two hardware platforms? Make sure then the software is the same. You need to outline the workloads that will be used for evaluation. What are the inputs in this system? Are you going to be able to run data that resembles what's seen in the real world or are you going to generate some synthetic traces; these are all important questions you need to resolve. Not to forget the metrics we talked about earlier in this lesson, is it execution time, or throughput, or response time, what is it that you care for and who are you designing the system for, is it the manager, is it the the resource usage in the system, or is it ultimately the customers?



So lets see now how these questions were treated in the Flash paper. Lets see what were the systems that they were comparing, what were the comparison points. First thing, the comparison with a MP version of the same kind of Flash processing, so a web server with the exact same optimizations that were applied in Flash, however, in a MP/single threaded configuration. Then again using the same optimizations as Flash, they put together a MTed web server that follows the Boss-Worker model. Then they compare Flash with a single process event-driven (SPED) model, so this is like the basic event-driven model that we discussed first. And then they also use as a comparison, two existing web server implementations, one was a

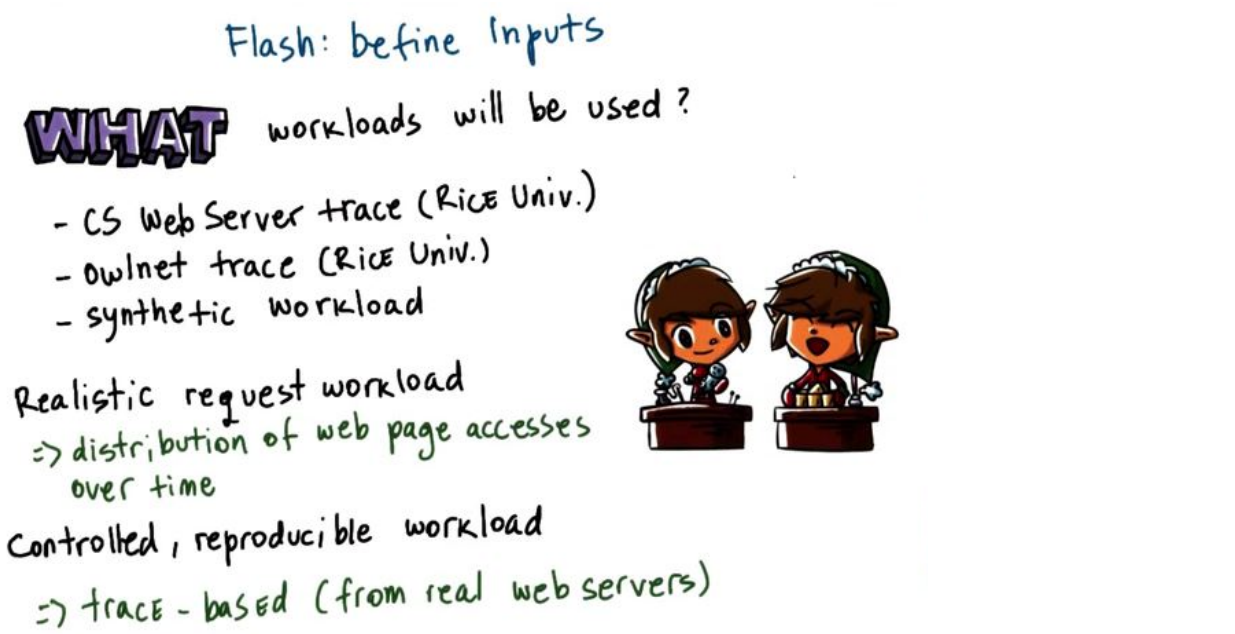
more research implementation that followed the SPED model, however, it used 2 processes and this was to deal with the blocking I/O situation, and then another one was Apache and this is the open-source Apache web server and this was at the time when this was done, this was a older version, obviously, than what's available today, at the time Apache was a MP configuration.



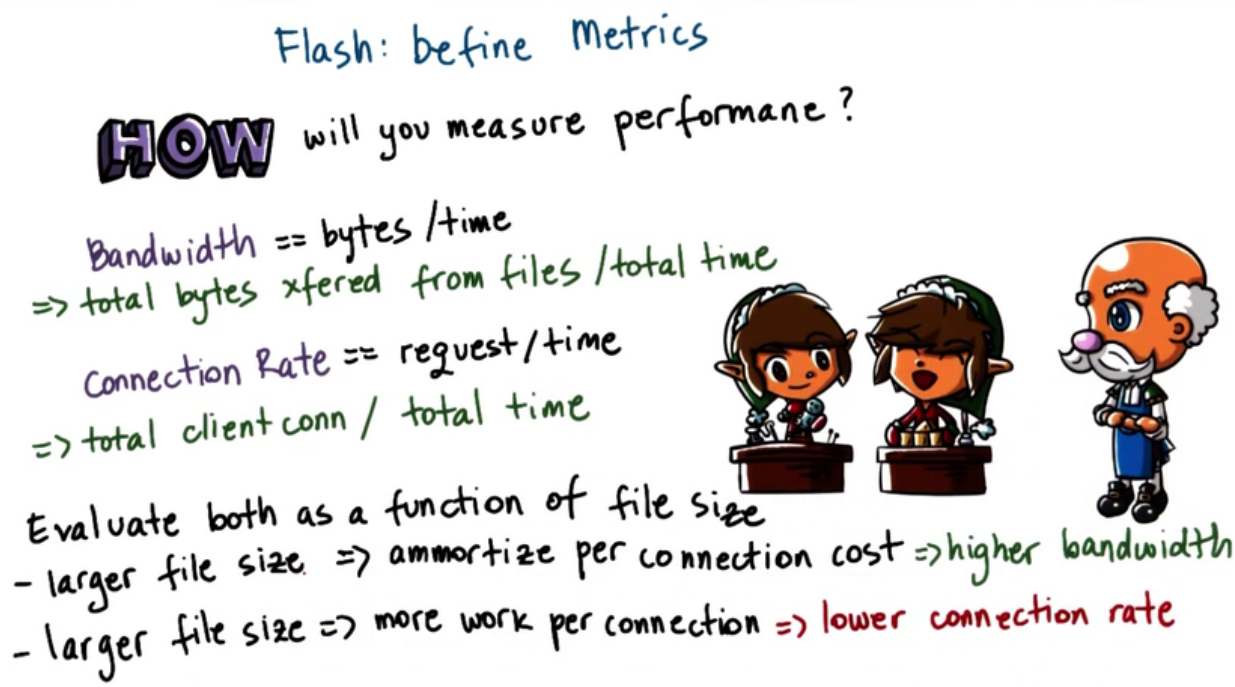
Except for Apache, every single one of these implementations integrated some of the optimizations that Flash already introduced. And then every single one of these implementations is compared against Flash, so this basically means is that we are comparing the different models, MP, MT, SPED, against the AMPED model, given that all of these really implement otherwise the exact same code with the same optimizations.

Next lets see what are the workloads they choose to use for the evaluations. To define useful inputs they wanted workloads that represent a realistic sequence of requests because thats what will capture a distribution of web page accesses, but they wanted to be able to reproduce, to repeat the experiment, with this same pattern of accesses. Therefore, they used a trace-based approach where they gathered traces from real web servers and then they replayed those traces so as to be able to repeat the experiment with the different implementation, so that every single one of the implementations can be evaluated against the same trace.

What they ended up with were two real-world traces, they were both gathered at Rice University where the authors are from, actually were from some of them are no longer there. The first one was the CS Web trace and the second one was the so called Owlnet trace. The CS trace represents the Rice University web server for the Computer Science Department and it includes a large number of files and doesn't really fit in memory. The Owlnet trace, that one was from a web server that hosted a number of student web pages and it was much smaller, so it would typically fit in the memory of a common server.



In addition to these two traces, they also used a synthetic workload generator, and with the synthetic workload generator, as opposed to replaying these traces of real-world page access distributions, they would perform some best or worst type of analysis or run some what-if questions, like what-if the distribution of the web page accesses had a certain pattern, would something change about their observations.



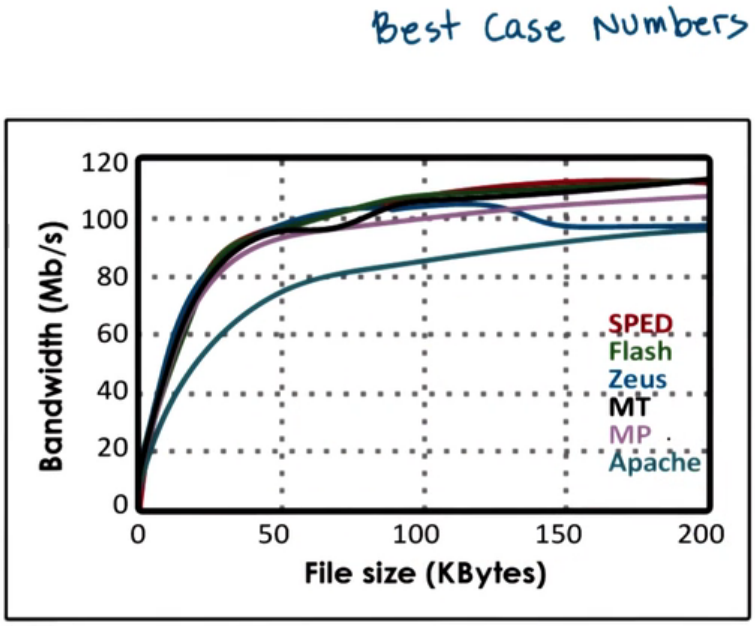
And finally, lets look at what are the relevant metrics that the authors picked in order to perform their comparisons. First, when we talk about web servers, a common metric is clearly bandwidth. So this is what is the total amount of useful bytes, so the bytes transferred from files over the time that it took to make that transfer, and the units is clearly bytes per second or

megabytes per second or similar. Second, because they were particularly concerned with Flash's ability to deal with concurrent processing they wanted to see the impact on connection rate as a metric. And that was defined as the total number of client connections that are services over a period of time.

Both of these metrics were evaluated as a function of the file size, so the understanding they were trying to gain was how does the workload property of requests that are made for different file sizes impact either one of these metrics. The intuition is that with a larger file size the connection cost can be amortized and that you can at this same time push out more bytes so you can, basically, obtain higher bandwidth. However, at the same time the larger the file, the more work that the server will have to do for each connection because it will have to read and send out more bytes from that larger file, so that will potentially negatively impact the connection rate. So this is why the chose that file size was a useful parameter to vary, and then understand its impact on these metrics for the different implementations.

22 - Experimental Results

Lets now look at the experimental results; we will start with the best case numbers. To gather the best case numbers, they use the synthetic load in which they varied the number of requests that are issued against the web server and every single one of the requests is for the exact same file. Like for instance, every single one of the requests is trying to get index.html. This is the best case because really in reality, clients will likely be asking for different files and in this pathological best case its likely, basically, the file will be in cache so everyone of these requests will be serviced as fast as possible. They're definitely wouldn't be any need for any kind of disk I/O.

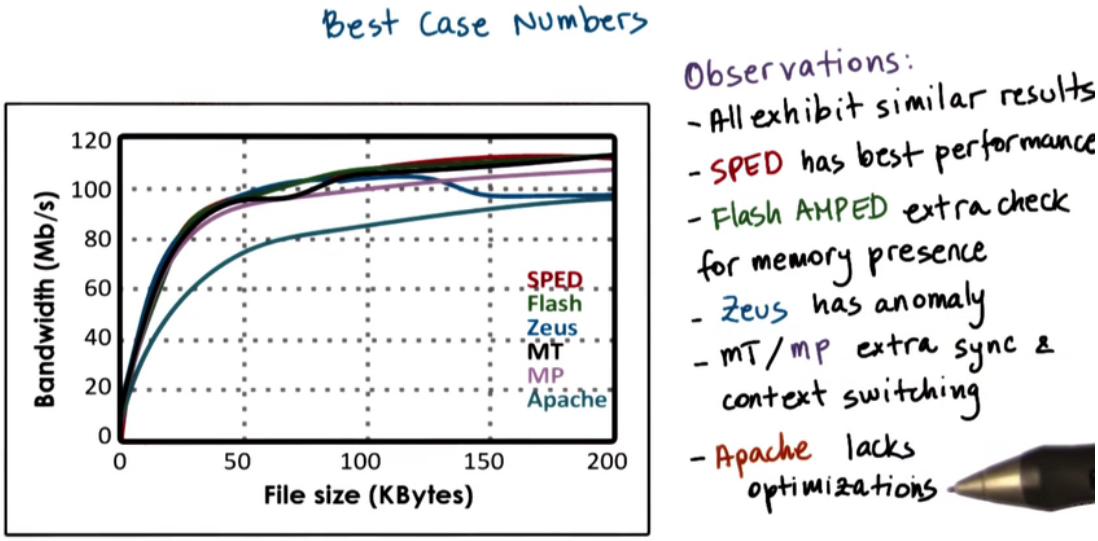


So for the best case experiments they measure bandwidth and they do that, they vary the file size of 0 to 200 KB and they measure bandwidth as the N, number of requests, times the file size over the time (BW = N * bytes(F) / time) that it takes to process the N number of requests

for this file. By varying the file size, they varied the work that both the web server performs on each request, but also the amount of bytes that are generated on a request. So we sort of assume that as we increase the filesize that the bandwidth will start increasing.

So lets look at the results now. The results show that the curves for every one of the cases that they compare. The Flash results are the green bar, SPED is the Single Process Event-Driven model, MT Multi-Threaded, MP Multi-Process, Apache this bottom curve corresponds to the Apache implementation. And Zeus that corresponds to the darker blue, this is the SPED model that has 2 instances of SPED, so the dual process event-driven model.

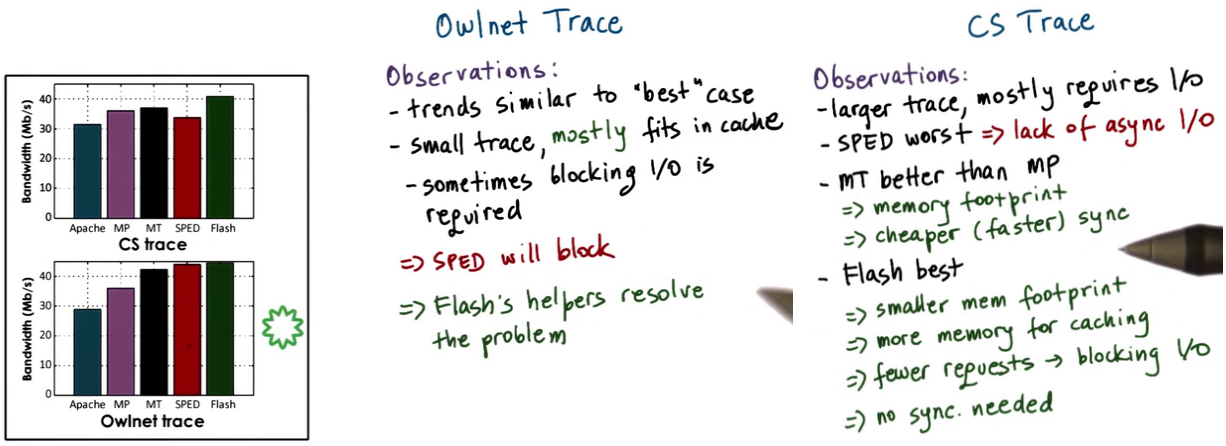
We can make the following observations. First, for all of the curves, initially when the file size is small, bandwidth is low, and as the file size increases the bandwidth increases. We see that all of the implementations have very similar results. SPED is really the best, thats the single process event-driven, and thats expected because it doesn't have any threads or processes among which it needs to context switch. Flash is similar but it performs that extra check for the memory presence. In this case, because this is single file trace, so every single one of the requests is for the single file, there's no need for blocking I/O, so none of the helper processes will be invoked, but nonetheless this check is performed so thats why we see a little bit lower performance for Flash.



Zeus has an anomaly, its performance drops here a little bit, and that has to do with some misalignment for some of the DMA operations. So not all of the optimizations are bug proof in the Zeus implementation. For the MT and the MP models, the performance is slower because of the context switching and extra synchronization. And the performance of Apache is the worst because it doesn't have any optimizations that the others implement.

Now since real clients don't behave like this synthetic workload, we need to look at what happens with some of the realistic traces, the Owlnet and the CS trace. Lets take a look at the Owlnet trace first.

First we see that for the Owlnet trace the performance is very similar to the best case, with SPED and Flash being the best, and then MT, MP, and Apache dropping down. Note what we're not including the Zeus performance. The reason for this trend is because the Owlnet trace is the small trace, though most of it will fit in the cache and we'll have a similar behavior like what we had in the "best" case where all the requests are serviced from the cache. Sometimes, however, blocking I/O is required, its mostly fits in the cache.



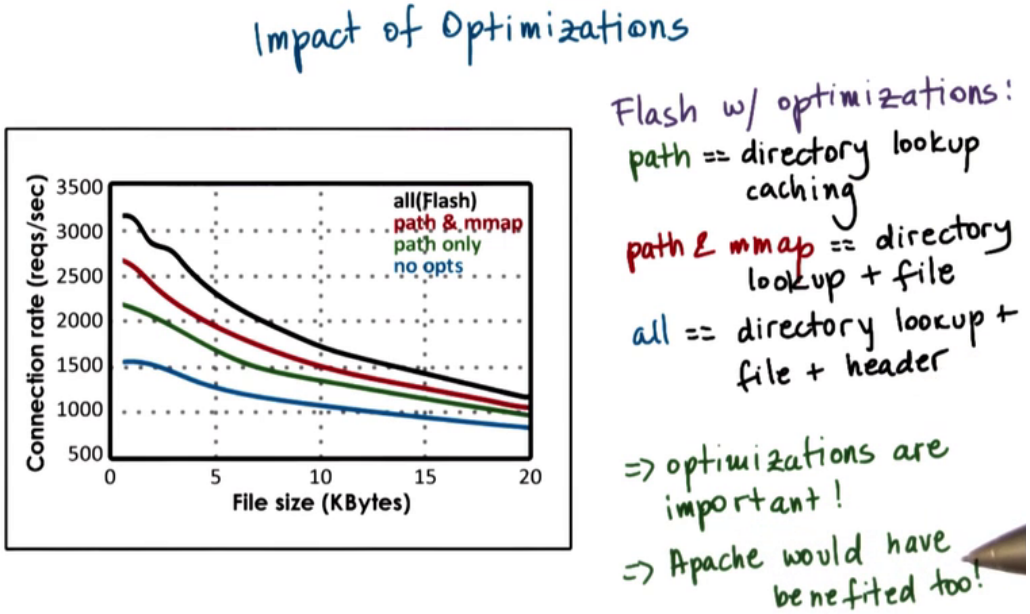
Given this, given the blocking I/O possibility SPED will occasionally block, whereas in Flash the helper processes will resolve the problem. And thats why we see here that the performance of Flash is slightly higher than the performance of the SPED. Now if we take a look at what's happening with the CS trace, this remember is a larger trace, so will mostly require I/O, its not going to fit in the cache, in memory in the system. Since the system does not support asynchronous I/O operations, the performance of SPED will drop significantly. So relative to where it was close to Flash [in the Owlnet trace], now its significantly below Flash and in fact its below the MP and the MTed implementations.

Considering the MTed and the MP, we see that the MTed is better than the MP, and the main reason for that is that the MT implementation has a smaller memory footprint. The smaller memory footprint means that there will be more memory available to cache files in turn that will lead to less I/O, so this is a better implementation. In addition, the synchronization and coordination and context switching between threads in a MTed implementation is cheaper, it happens faster, than among processes in a MP implementation.

In all cases, Flash performs best, then it has the smaller memory footprint compared to the MTed and the MP, and that results in more memory available for caching files or caching headers. As a result of that, fewer requests will lead to blocking I/O operation which further speeds things up. And finally given that everything happens in the same address space, there isn't a need for explicit synchronization like with the MTed or MP model. And this is what makes Flash perform best in this case.

In both of those cases, Apache performed worst, so lets try to understand if there's really an impact of the optimizations performed in Flash. And here are the results represent the different

optimizations, the performance thats gathered with Flash without any optimizations, thats the bottom line. Then Flash with the path only optimizations, so the path only thats the directory lookup caching, so thats like the computation caching part. Then the red line here, the path and mmaps, so this includes caching of the directory lookup plus caching of the file. And then the final bar or the final line, the black line, that includes all of the optimizations, so this is the directory lookup, the file caching, as well as the header computations of the file.

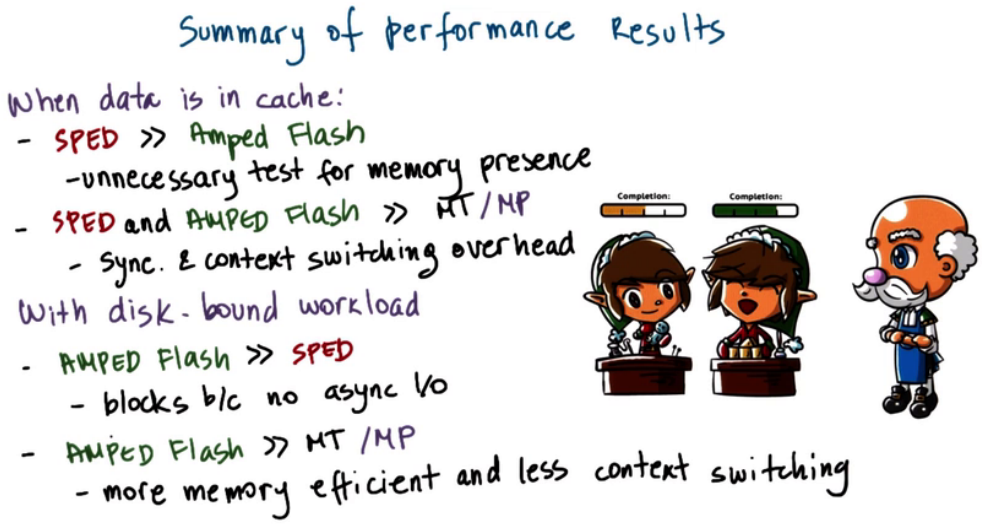


And we see that as we add some of the optimizations, this impacts the connection rates, so the performance that can be achieved by the web server significantly improves, we're able to sustain a higher connection rate as we add these optimizations. This tells us two things, first that these optimizations are indeed very important, and second they tell us that the performance of Apache would have been also impacted if it had integrated some of these same optimizations as the other implementations.

23 - Summary of Performance Results

To summarize, the performance results for Flash show the following. When the data is in cache, the basic SPED model performs much better than the AMPED Flash because it doesn't require the test for memory presence which was necessary in the AMPED Flash. Both SPED and the AMPED Flash are better than the MTed or MP models because they don't incur any of the synchronization or context switching overheads that are necessary with these (MT/MP) models.

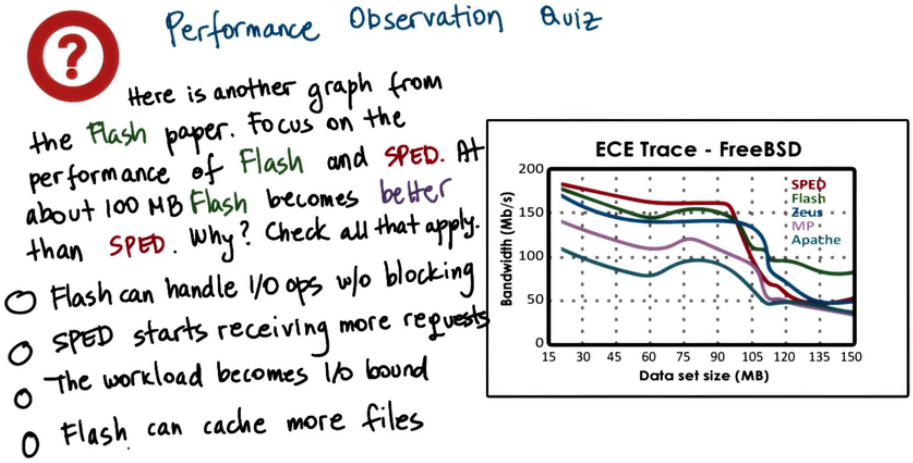
When the the workload is disk-bound, however, AMPED performs much better than the single process event-driven (SPED) model because the single process model blocks since there's no support for asynchronous I/O. AMPED Flash performs better than both the MTed and the MP model because it has much more memory efficient implementation and it doesn't require the same level of context switching as in these (MT/MP) models. Again, only the number of concurrent I/O bound requests result in concurrent processes or concurrent threads in this model.



The model is not necessarily suitable for every single type of server process, there are certain challenges with event-driven architecture, we said some of these can come from the fact that we need to take advantage of multiple cores and we need to be able to route events to the appropriate core, in other cases perhaps the processing itself is not as suitable for this type of architecture. But if you look at some of the high performance server implementations that are in use today you will see that a lot of them do in fact use a event-driven model combined with asynchronous I/O support.

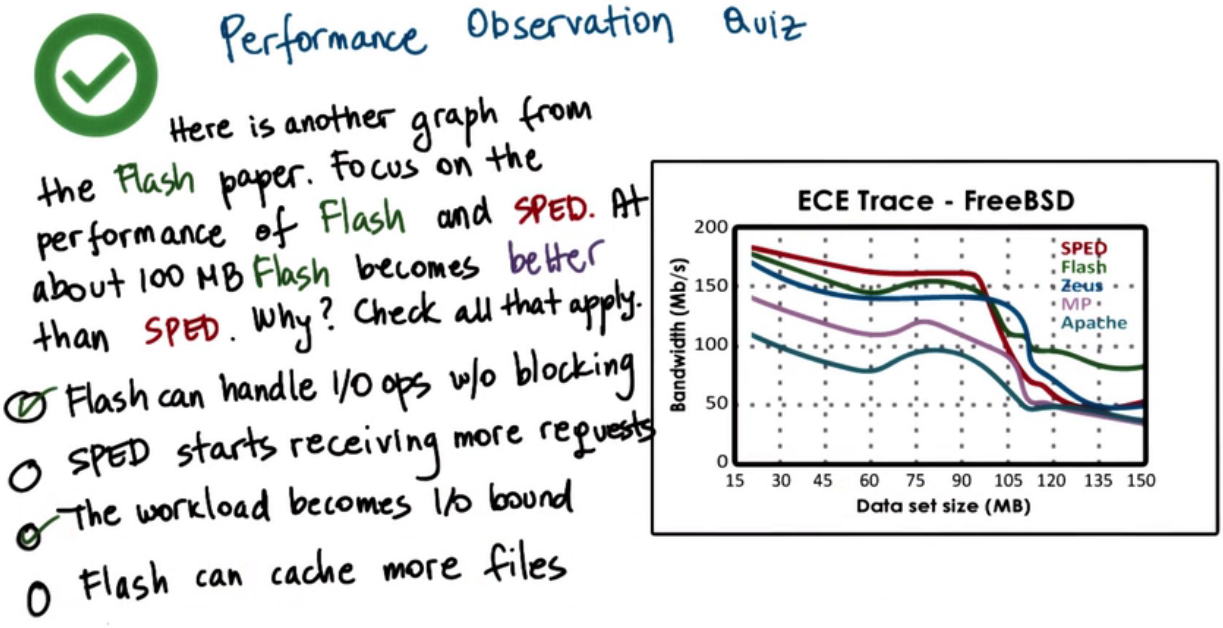
24 - Performance Observation Quiz

Lets take one last look at the experimental results from the Flash paper as a quiz this time. Here is another graph from the Flash paper, and focus on the green and the read bars that correspond to the SPED model and the Flash AMPED model. You see that at about 100 MB the performance of Flash becomes better than the performance of the SPED model. Explain why, and you should check all that apply. From the answers below. Flash can handle I/O operations without blocking. At that particular time, SPED starts receiving more requests. The workload becomes I/O bound. Or Flash can cache more files.



25 - Performance Observation Quiz

The first answer is correct, yes Flash has the helper processes so it can handle I/O operations without blocking. The second answer really makes no sense; both processes continue receiving the same number of requests in these experiments. The third answer is correct as well, at 100 MB the workload, its size increases, it cannot fit in the cache as much as before and so it becomes more I/O bound, there are more I/O requests that are needed beyond this point.



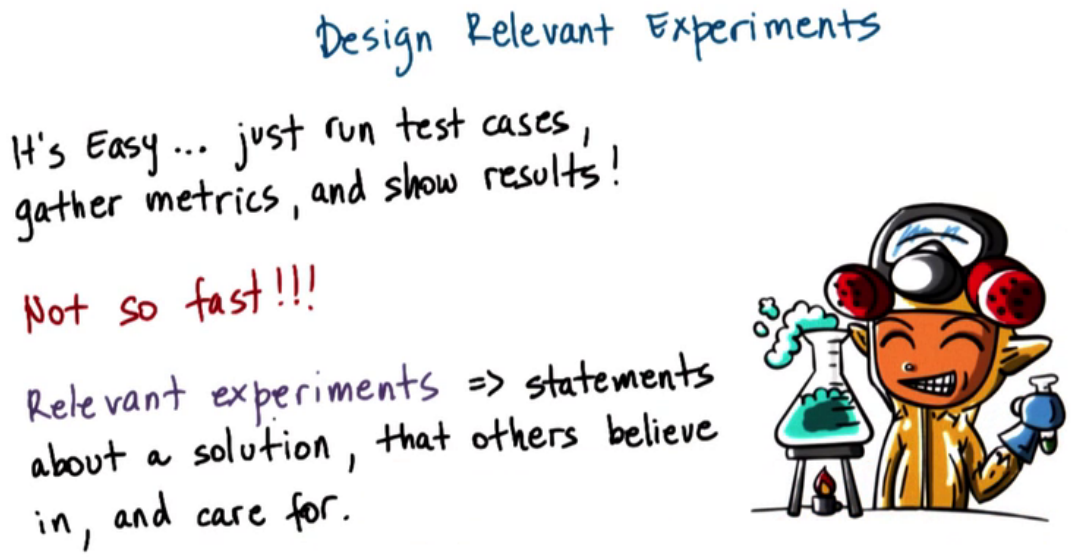
For SPED, at this point (100 MB) once the workload starts becoming more I/O bound the problem is that a single blocking I/O operation will block the entire process. None of the other requests can make progress and that's why its performance significantly drops at that point. And finally the last answer, that Flash can handle more files, that's really not correct; SPED and Flash have comparable memory footprints and so it is not that one can handle more files than the other in the memory cache. If anything Flash has the helper processes, so if those are created, they're going to interfere with the other available memory and will impact the number of available cache in the negative sense, so if anything it will have less available memory for caching files than SPED, so this is not answer that explains why the Flash performance is better than the SPED performance.

26 - Advice on Designing Experiments

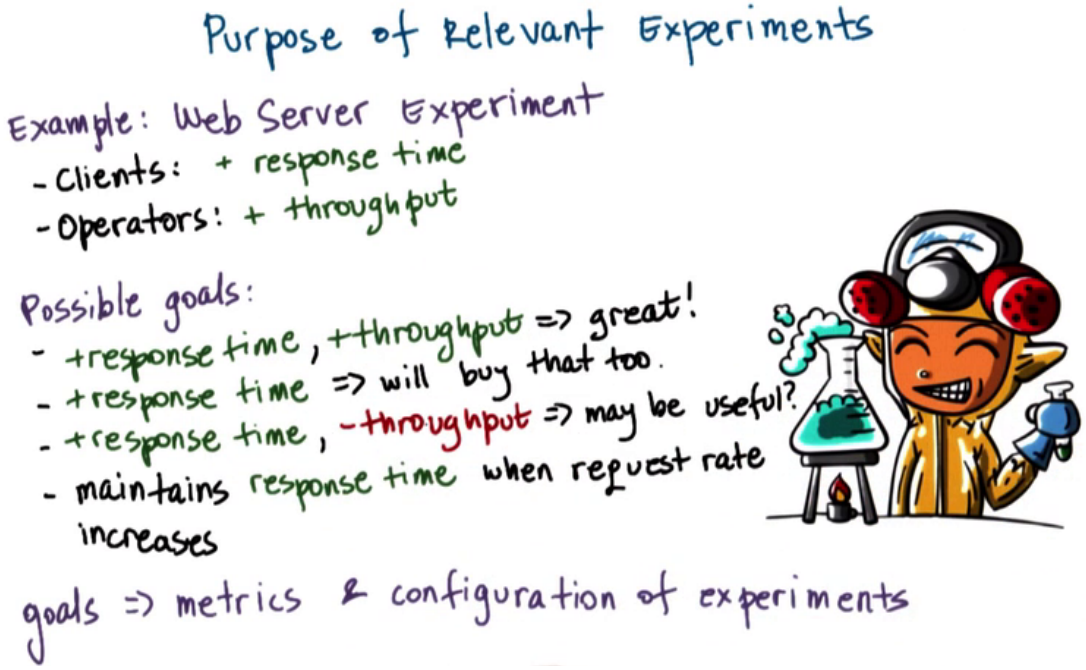
Before we conclude this lesson, I'd like to spend a little more time to talk about designing experiments. It sounds like its easy, we just need to run a bunch of test cases, gather the metrics, and show the results! NOT SO FAST, actually! Running tests, gathering metrics, and plotting the results is not as straightforward as it might seem. There is actually a lot of thought and planning that should go into designing relevant experiments.

By ***relevant experiment*** I'm referring to an experiment that will lead to certain statements about a solution, that are credible, that others will believe in, and that are also relevant, that they will

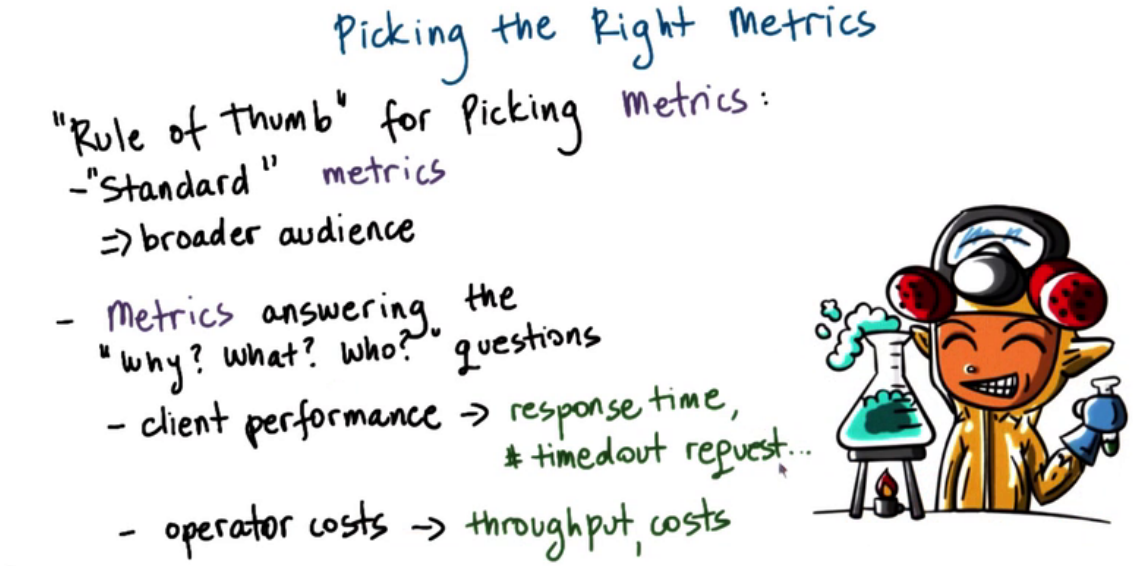
care for. For example, the paper we discussed is full of relevant experiments. There the authors provided the detailed descriptions of each of the experiments so that we could understand them and then we could believe that those results are sane. And then we were also able to make well founded statements about Flash and the AMPED model versus all of the other implementations.



Lets continue talking about the web server as an example for which we'll try to justify what makes some experiments relevant. Well the clients using the web server, they care for the response time, how quickly do they get a web page back. The operators, for instance, running that web server or that web site, they care about throughput, how many total client requests can see that web page over a period of time. So this illustrates that you will likely need to justify your solution using some criteria that's relevant to the stakeholders.



For instance, if you can show that your solution improves both response time and throughput, everybody's positively impacted, so that's great! If you can show that your solution only improves response time but doesn't really affect throughput, well okay I'll buy that too. It serves me some benefit. If I see a solution that improves response time and actually degrades throughput, that still could be useful. Perhaps for this improved response time, I can end up charging clients more ultimately will give the revenue that I'm losing due to the negative throughput. Or maybe I need to define some experiments in which I'm trying to understand how is the response time that the clients see, how is it affected when the overload of the web server increases, when the request rate increases. So by understanding the stakeholders and the goals that I want to meet, with respect to these stakeholders, I'm able to define what are some metrics that I need to pay attention to, and that will give me insights into useful configurations of the experiments.

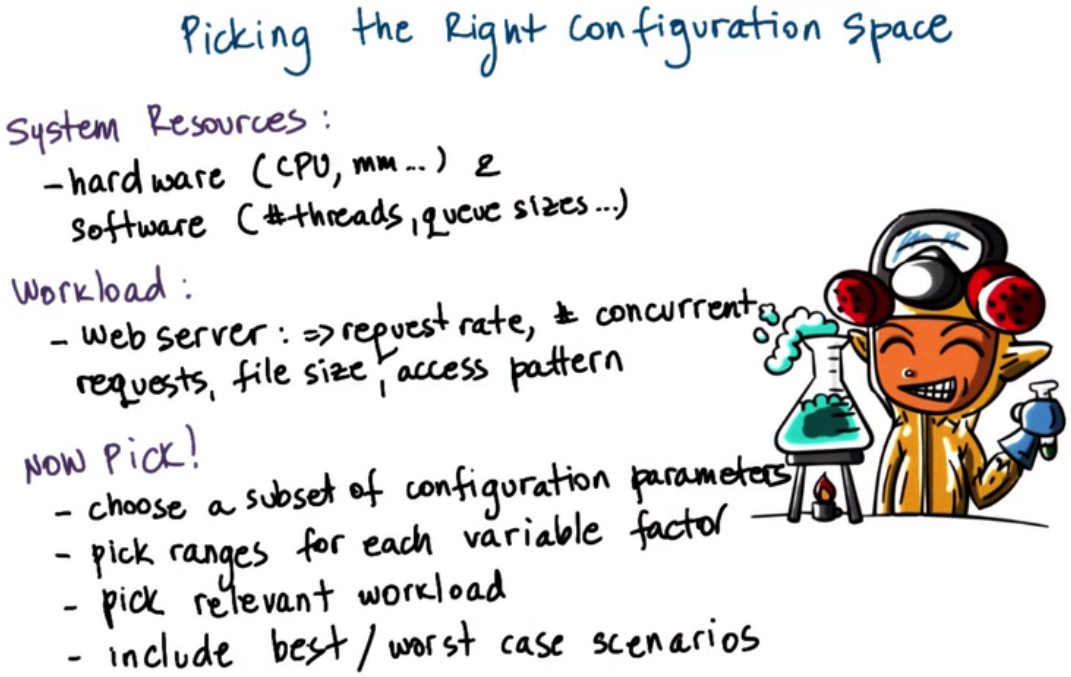


When you're picking metrics, a rule of thumb should be, what are some of the "standard" metrics that are popular in the target domain? For instance, for web servers it makes sense to talk about the client request rate, or the client response time. This will let you have a broader audience, more people will be able to understand the results and to relate to them, even if those particular results don't give you the best punch line.

Then you absolutely have to include metrics that really provide answers to questions such as: "Why am I doing this work?" "What is it that I want to improve or understand by doing these experiments?" "Who is it that cares for this?" Answering these questions implies what are the metrics that you need to keep track of. For instance, if you're interested in client performance, probably the things that you need to keep track of are things like response time, or number of requests that have timed-out. Or if you're interested in improving the operator costs then you worry about things like throughput or power costs, and similar.

Once you understand the relevant metrics you need to think about the system factors that affect those metrics. One aspect will be things like system resources. This will include hardware

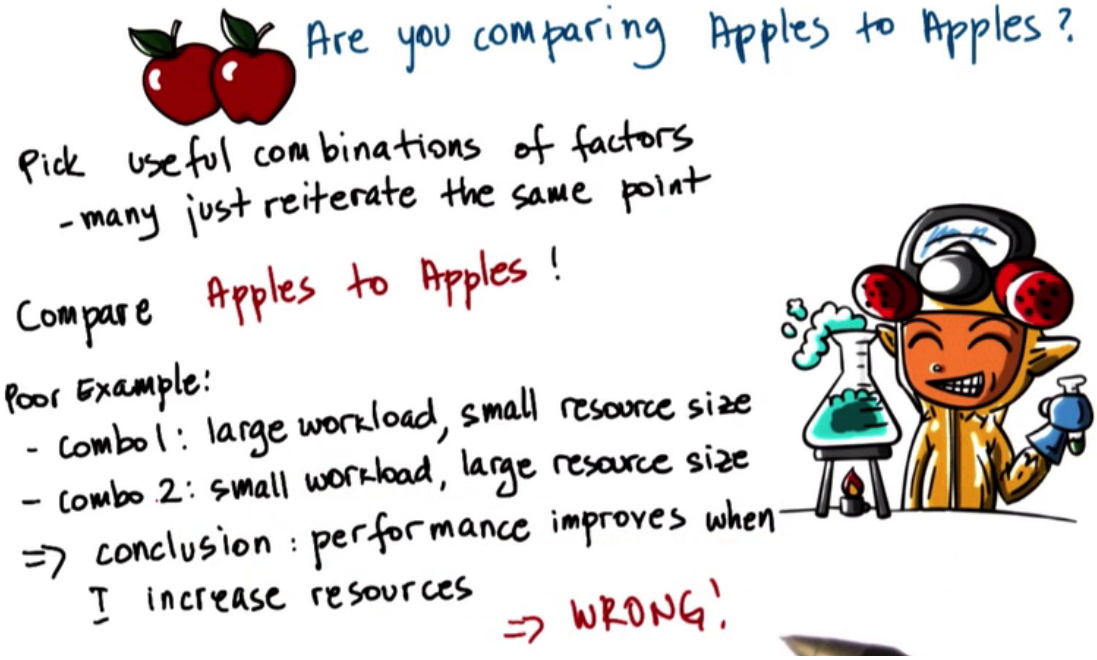
resources such as number and type of CPUs, or amount of memory that's available on the server machines. And also the software specific resources, like number of threads, or the size of certain queues or buffer structures that are available in the program. Then there are a number of configuration parameters that define the workload. Things that make sense for a web server include they're request rate, the file size, the access pattern, things that are varied also in the Flash experiments.



And now that you understand the configuration space a little better, make some choices! Choose some subset of the configuration parameters that probably are most impactful when it comes to changes in the metrics that you're observing. Pick some ranges for these variable parameters. These ranges must also be relevant. Don't show that your server runs well with 1, 2, and 3 threads, so don't vary the number of threads in your server configuration if you look out and see that real-world deployments, they have servers with thread counts in the hundreds. Or don't go and vary the file sizes to have sizes of 10B, 100B, and 1KB if you look at what's happening in the real-world, file sizes range from 10's of bytes up to 10's of megabytes and 100's of megabytes and beyond. So make sure that the range are representative of reality.

Again these ranges must some how correspond to some realistic scenario that relevant, otherwise nobody will care for your hypothetical results. That is unless your hypothetical results are concerned with demonstrating the best or the worst case scenarios. Best and worst case scenarios do bring some value because they in a way demonstrate certain limitations, or certain opportunities that are there because of the system that you've proposed, because of the solution you have proposed. So these are the only times where picking a non-realistic workload makes sense. Like for instance, in the Flash paper case, they had an example in which every single one of the requests was accessing one single file and there was some value results that were obtained through that experiment.

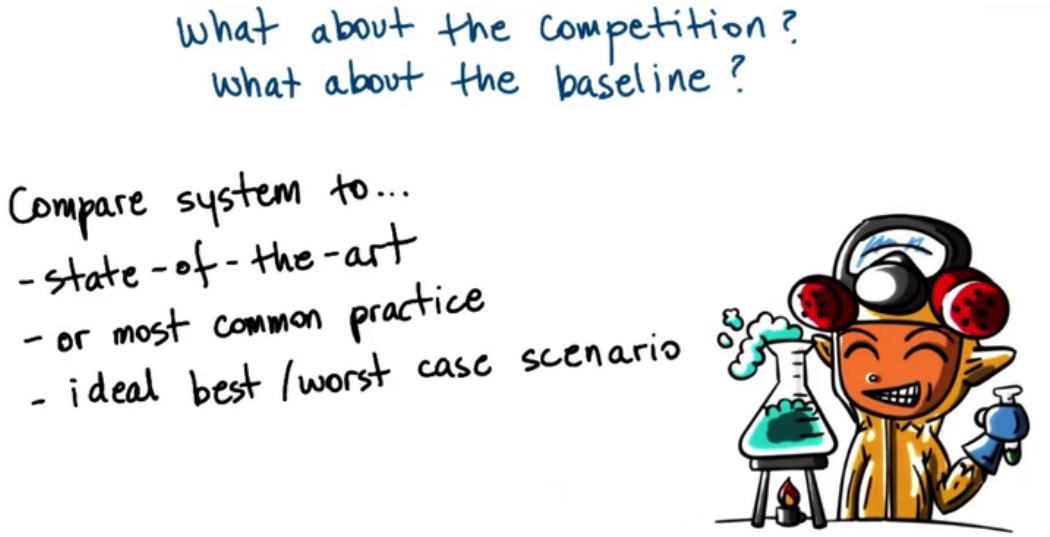
For the various factors that you are considering pick some useful combinations. There will be a lot of experiments where the results simply reiterate the same point. It really doesn't make sense to make endless such results. Few are good, its good to confirm that some observation is valid, but including 10's of them it really doesn't make any sense. A very important point, compare apples to apples!



For instance, lets look at one bad example. We have one combination in which we run an experiment with a large workload, and a small size of resources. And a second experiment, a second run of the experiment in which we've change the workload, so now we have a small workload and then we have also allocated more resources for it, so for instance more threads. And then we look at these results and we see that in the second case, so for the second experimental run, the performance is better so then we may draw conclusion, oh well I've increased the resource size, I've added more threads, and therefore my performance is improved. So I must be able to conclude that performance improves when I increase the resources. That's clearly wrong! I have no idea whether performance improved because I've added more resources, or because I have changed the workload, so I'm adding, I'm using a much smaller workload in the second case. This is what we mean by make sure that you are comparing apples to apples. There's no way that you can draw conclusion between these two experiments.

And what about the competition? What is the baseline for the system that you are evaluating? You should think about experiments that are able to demonstrate that the system you are designing, the solution you are proposing, in some way improves the state-of-the-art, otherwise it's not clear why use yours. And if it's not really the state-of-the-art when at least what's the most common practice, that should be improved. And perhaps there's some other benefits over the state-of-the-art that are valuable. Or at least think about evaluating your system by

comparing with some extreme conditions in terms of the workload or resource assignment, so some of the best or worst case scenarios. That will provide insights into some properties of your solution, like how does it scale as the workload increases, for instance.



27 - Advice on Running Experiments

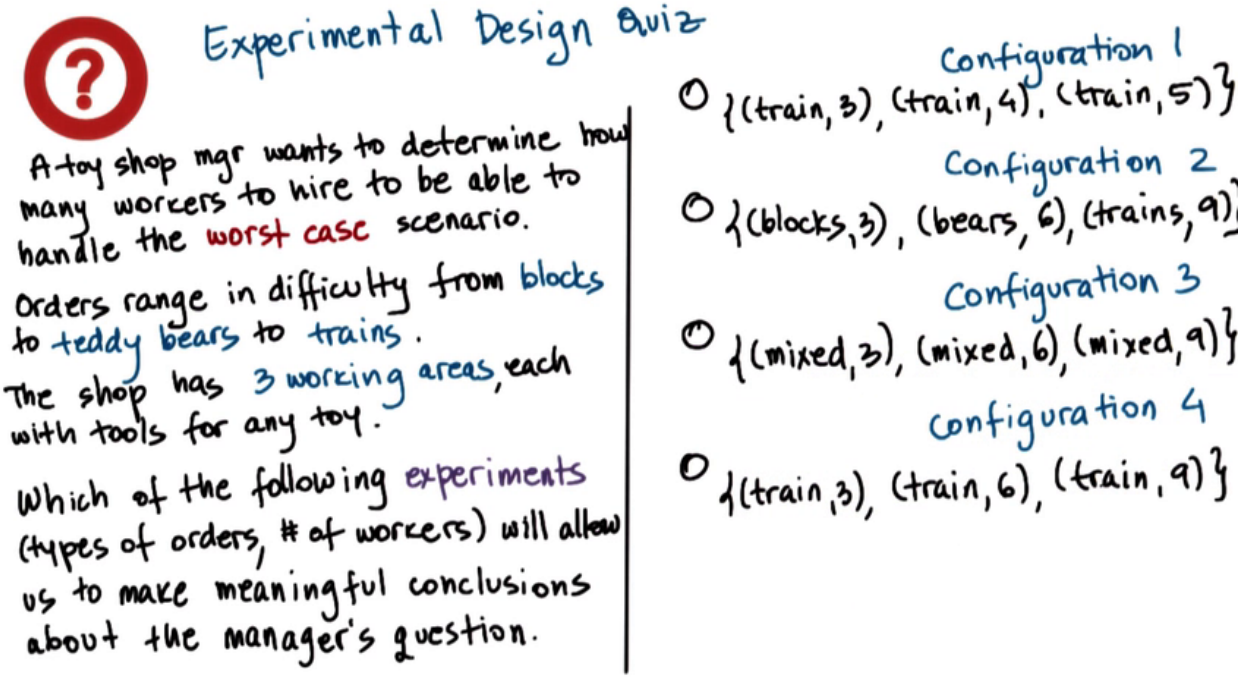
Okay, at this point we have designed the experiments and now what? And now it actually becomes easy, now that you have the experiments, you need to run the test cases a n number of times using the different ranges of the experimental factors, compute the metrics (the averages over those n times), and then represent the results.

When it comes to representing the results, I'm not going to go into further discussion, in terms of best practices on how to do that, but just keep in mind that the visual representation can really

help strengthen your arguments. And there are a lot of papers that will be discussed during this course that use different techniques on how to represent results so you can draw some ideas from there. Or there are other documentations online, there are also courses that are taught at Georgia Tech or also on Udacity's platform that talk about information visualization. So you can benefit from such content in terms of how to really visualize your results. And make sure that you don't just show the results, actually make a conclusion, spell out what is it that these experimental results support as fair as your claims are concerned.

28 - Experimental Design Quiz

Lets now take a quiz in which we will look at a hypothetical experiment and will try to determine if the experiments we are planning to conduct will allow us to make meaningful conclusions. A toyshop manager wants to determine how many workers he should higher in order to be able to handle the worst case scenario in terms of orders that are coming into the shop. The orders range in difficulty starting from block, which are the simplest, to teddy bears, to trains, which are the most complex ones. The shop has three separate working areas, and in each working area there are tools that are necessary for any kind of toy. These working areas can be shared among multiple workers.

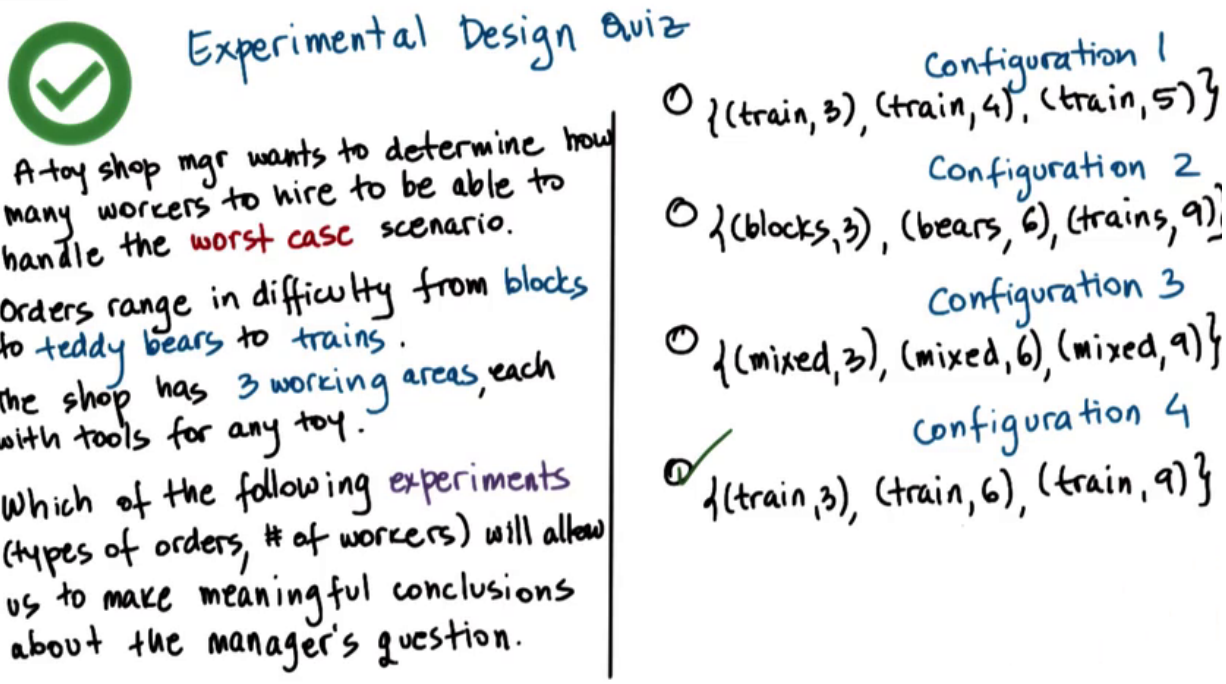


Which of the following experiments that are represented as a tuple of types of orders that being processed, and number of workers that processes this order, will allow us to make meaningful conclusions about the manager's question. The first configuration has three trials. In each trial, we use trains as the workload so order of trains, and we vary the number of workers, 3, 4, and 5. In the second configuration, again we have three trials. The first trial consists of order of blocks with three workers, the second trial is an order of bears with six workers, and the third trial is an order of trains with nine workers. In the third configuration, in each of the trials we have a mixed set of orders, so of all the different kinds, and we vary the number of workers from

3 to 6 to 9. And in the fourth configuration, in each of the trails we use a set of train orders and we vary the number of workers from 3 to 6 to 9

29 - Experimental Design Quiz

Lets quickly talk about what the toyshop manager should want to evaluate. It should be something like this. Given that the most complex case of toy orders includes trains then we should have in each of the trials, a set of orders that are really for trains. Second the toyshop has three working areas. We can perform any kind of toy order in each of the working areas and multiple workers can share an area. So as we're trying to see how many more workers can we add in the system, how many more toys can we process we really should be trying to get as many more workers per working area.



Now if we take a look at configuration 1, it has correctly in each trial an order of trains, so a sequence of train orders. That corresponds to our worst case scenario. However, the way the workers are varied, in the first case there are a total of three workers so there is one in each working areas. In the second case, there are a total of four workers so the first working area has one extra worker. So the number of resources in that case is larger for the first working area and lower for the next two. Similarly in the third trial, we have in two working areas two workers and in the last one just one. Its really hard to draw any conclusions, the amount amount of resources that's available in each of these [areas] for handling the toys is not equal, therefore it doesn't really tell us anything about the worst case capacity of the system.

If we take a look at the second configuration, here we have the first trial is an order of blocks, the second trial is an order of bears, the third trial is an order of trains. Again it doesn't tell us anything about the worst case capacity of the system. This could tell us something, but it really is not the question that the manager is asking. The third configuration similarly, it could provide

some information, in every single one of these the workload is mixed, so this could correspond to the average number of toys that can be processed with different number of workers, so how is the average throughput impacted by adding more workers in the store. Again, however, this doesn't address the question of how is the worst case impacted by adding more workers to the store.

So that basically gives us the answer to the final configuration, the last configuration is identical to configuration three in the number of workers, but it uses the worst case scenario, so its just orders of trains. So this tells us how much better will I be able to handle the worst case of just receiving trains if I add more and more workers. And I'm really adding an even amount of workers per working area.

This is a meaningful set of experiments, that will let me draw some conclusions. It will also likely ultimately demonstrate what is the capacity of the individual working areas. So lets say if I try maybe to add another trial where I'm running train orders with 12 workers, so 4 workers per working area. Likely I will, at some point, no longer start seeing any kind of improvement simply because I cannot squeeze in more workers per working area. So performing this type of experiment will actually be useful.

30 - Lesson Summary

In this lesson, we introduced the event-driven model for achieving concurrency in applications. We performed comparisons between multi-process (MP), multi-threaded (MT), and event-driven approach for implementing a web server application. And in addition, we discussed in more general terms how to properly structure experiments