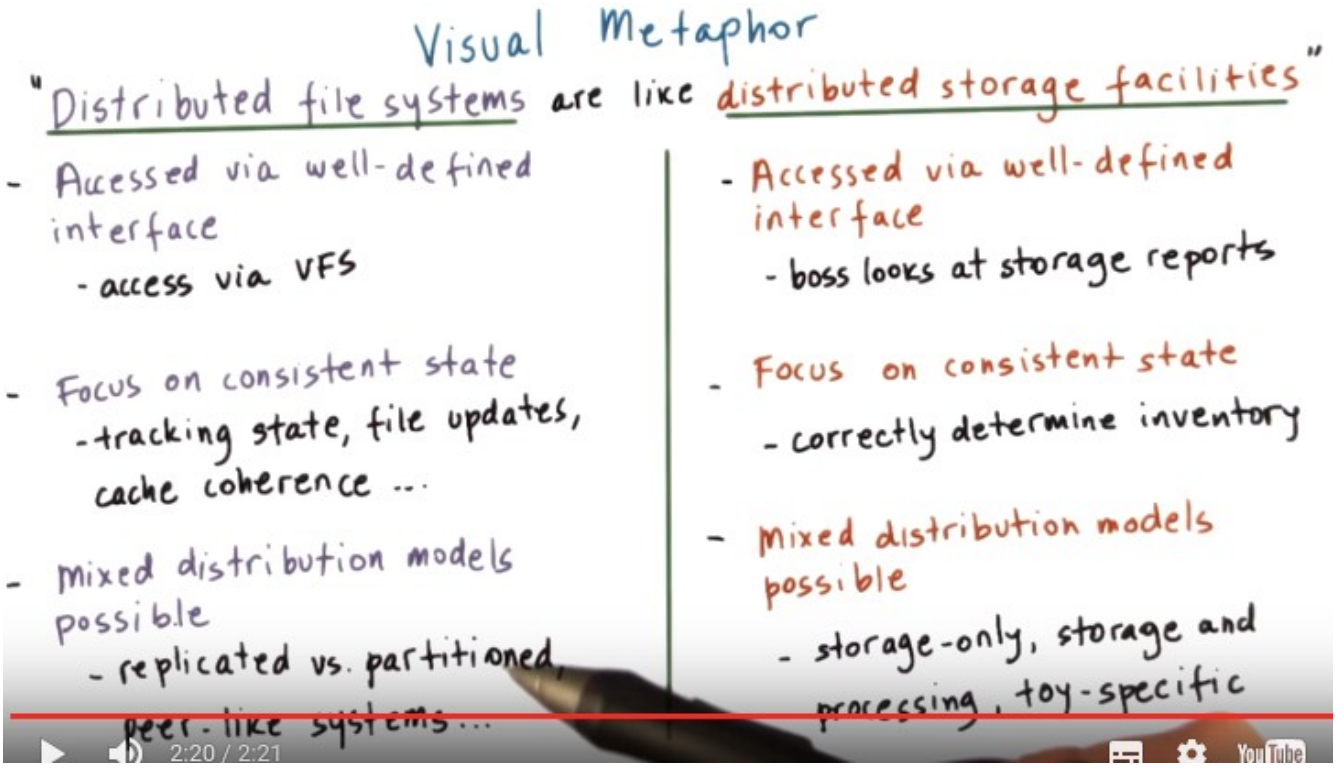
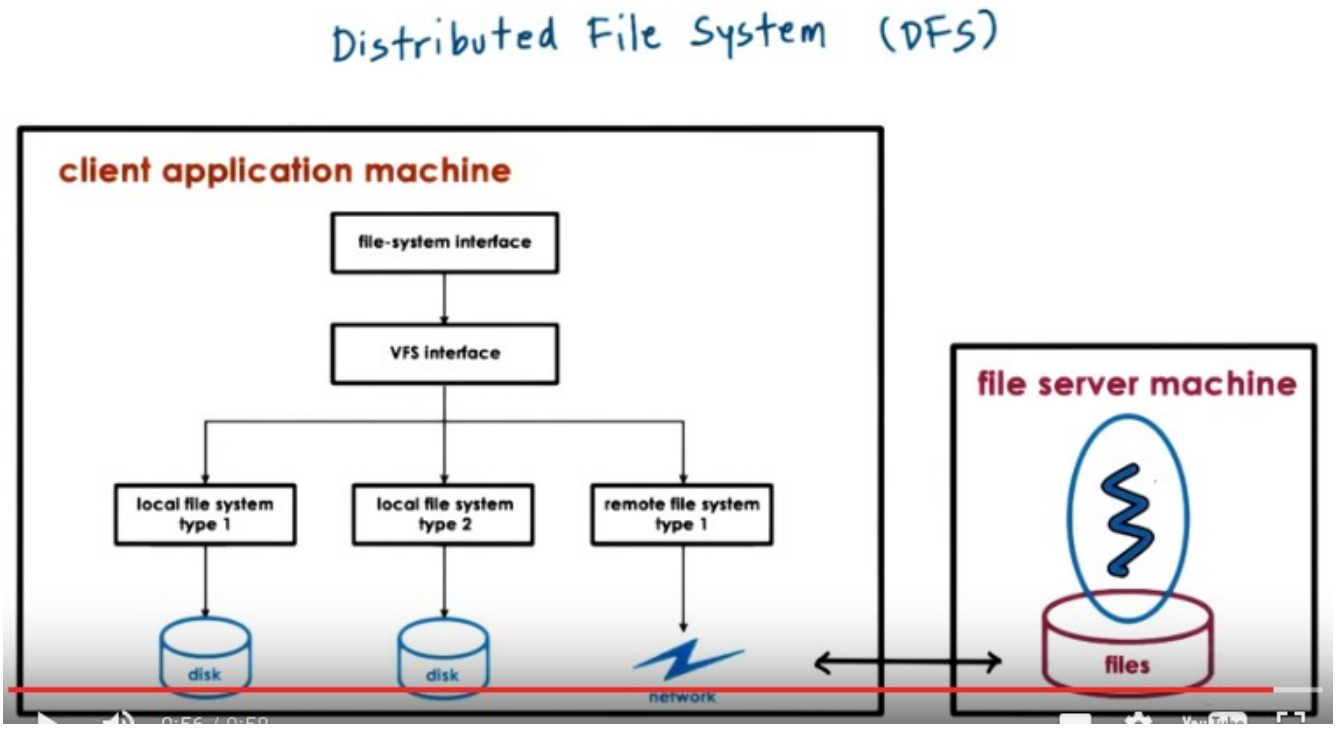
DFS: distributed file system

1. In this lesson we will talk about distributed file systems and about some of the design decisions that are available in this space. We will describe what are the specific decisions made for NFS, which is a popular example of a distributed file system. And we will also look at a research paper, Caching in the Sprite Network File System. This paper is useful because it has a lot of information why certain decisions were made in the Sprite file system. One thing you should have in mind is that all of the discussion in this lesson will focus on distributed file systems. The methods that we will talk about generalize to other types of distributed services.

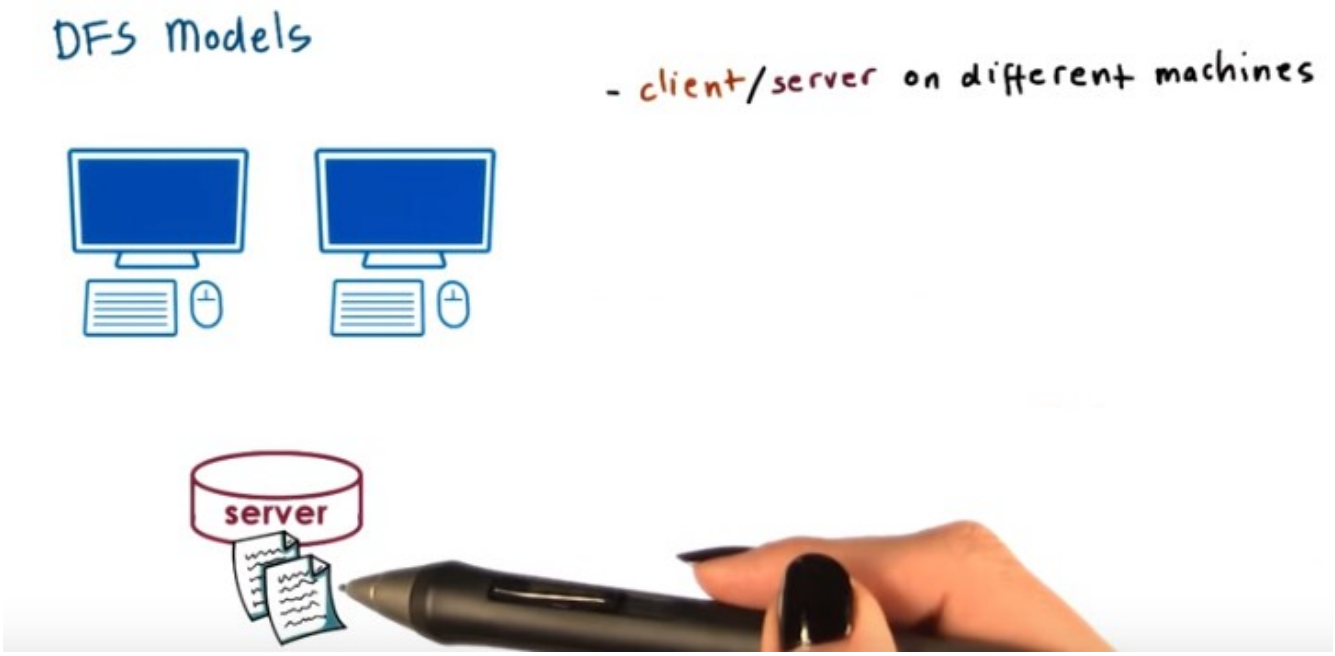


VFS: virtual file system

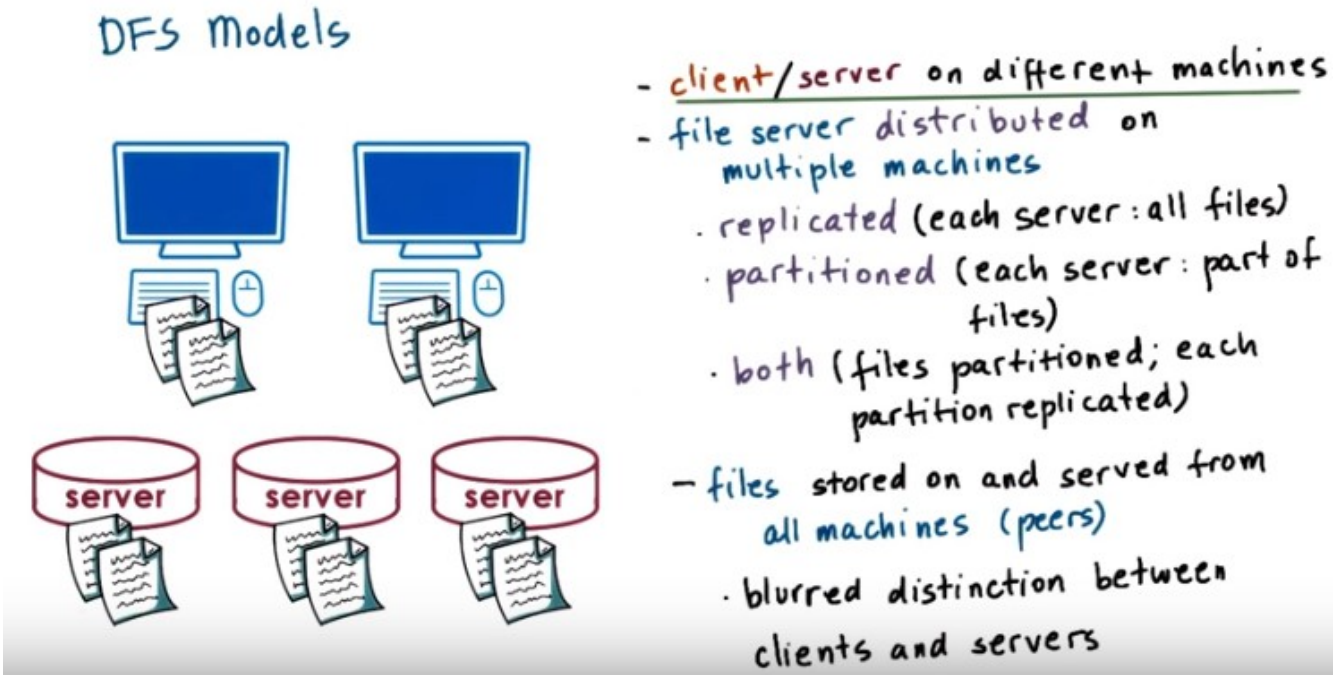
2. First, let's start with a visual metaphor. Distributive file systems are like distributed storage facilities, specifically storage facilities containing toy parts for a toy shop. First, each is accessed via some well-defined interface. Also, both focus on maintaining consistent state as one of the major functional requirements. And finally, both of these support different distribution models. For the distributed storage facilities, the toy shop manager needs to be able to access, check, and make decisions regarding the available resources without having to leave the toy shop and directly visit some of the storage facilities. This should be done via some well-defined interfaces that the storage facilities export. Next, the distributed storage facilities must constantly update their inventories to represent consistent information about the parts that they can deliver. This helps managers and other workers accurately determine what are the inventories and what are the times that these products can be delivered, for instance. Finally, these distributive facilities can be configured in different ways. Some of them can be for storage-only. Some can provide both storage and processing services. Some can be specialized for different types of toys or different types of parts. So there will be different types of distribution models. Now let's look at how distributed file systems relate to these properties of distributed storage facilities. First, distributed file systems are also accessed via via some higher level well-defined interfaces. In particular, this would be the virtual file system interface that we already talked about. This allows the operating system to take advantage of multiple types of storage devices or storage machines, regardless of where they are. Next, distributed file systems need to maintain consistent state of the files shared among clients. If a file is updated by one client, the distributed file system needs to track information regarding all of the file updates, some state that's necessary, for instance, for execution of cache coherence, algorithms, and other information. And also, the distributed file systems can be implemented via different distribution models. They can be replicated or the files can be partitioned across the front machines in the file system. Or all of the notes in the file system can act more like peers amongst each other when they're providing the distributed file system service.



3. >From our previous discussion about file systems, we said that modern operating systems hide the fact that there may be different types of storage devices with different types of internal file system organizations. And this is done by exporting a higher level virtual file system interface. Underneath this virtual file system interface, the operating system can also hide the fact that there isn't even local physical storage on a particular machine where the files are stored. But that instead everything is maintained on a remote machine, or on a remote file system that is being accessed over the network. These kinds of environments, where there are multiple machines that are involved in the delivery of the file system service together form a distributed file system. In this case, simply the client, where the data may be cached, and the server, where the data is stored, sit on two different machines, and that's enough for us to have a distributed file system.



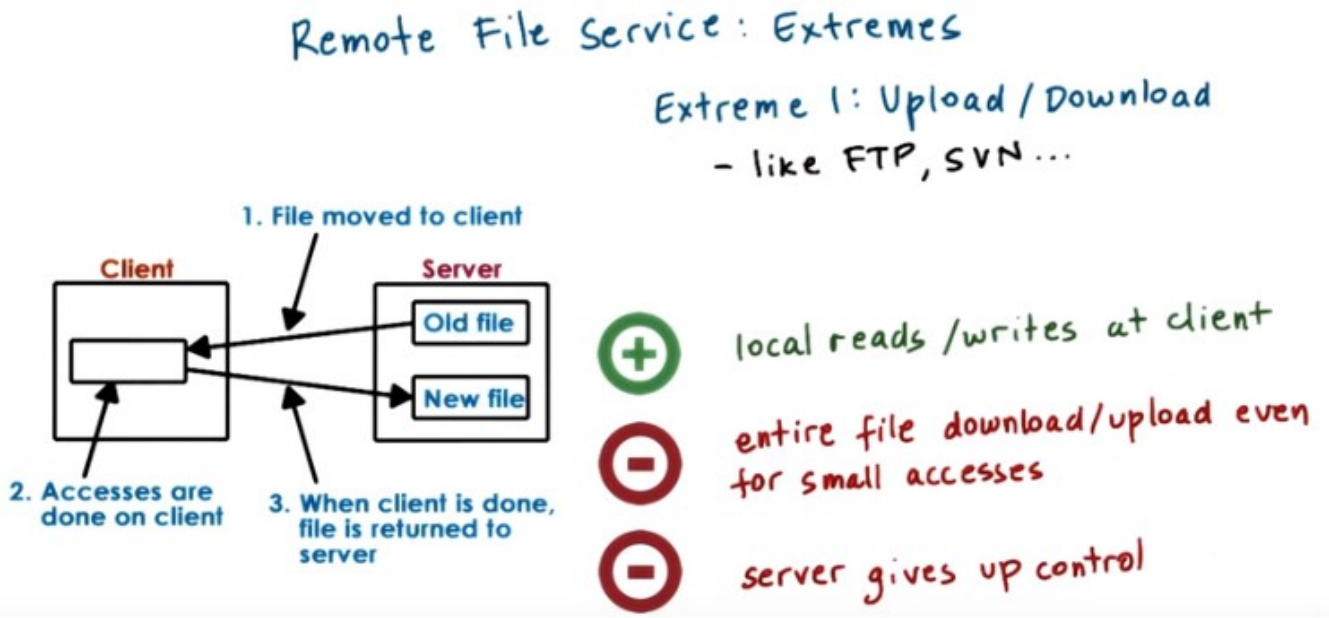
4. More generally, a distributed file system is a file system that can be organized in any of the following ways. The first model is when the client or the clients and the file server are on different machines.



注意 partitioned 是 part of files, 不是 part of file

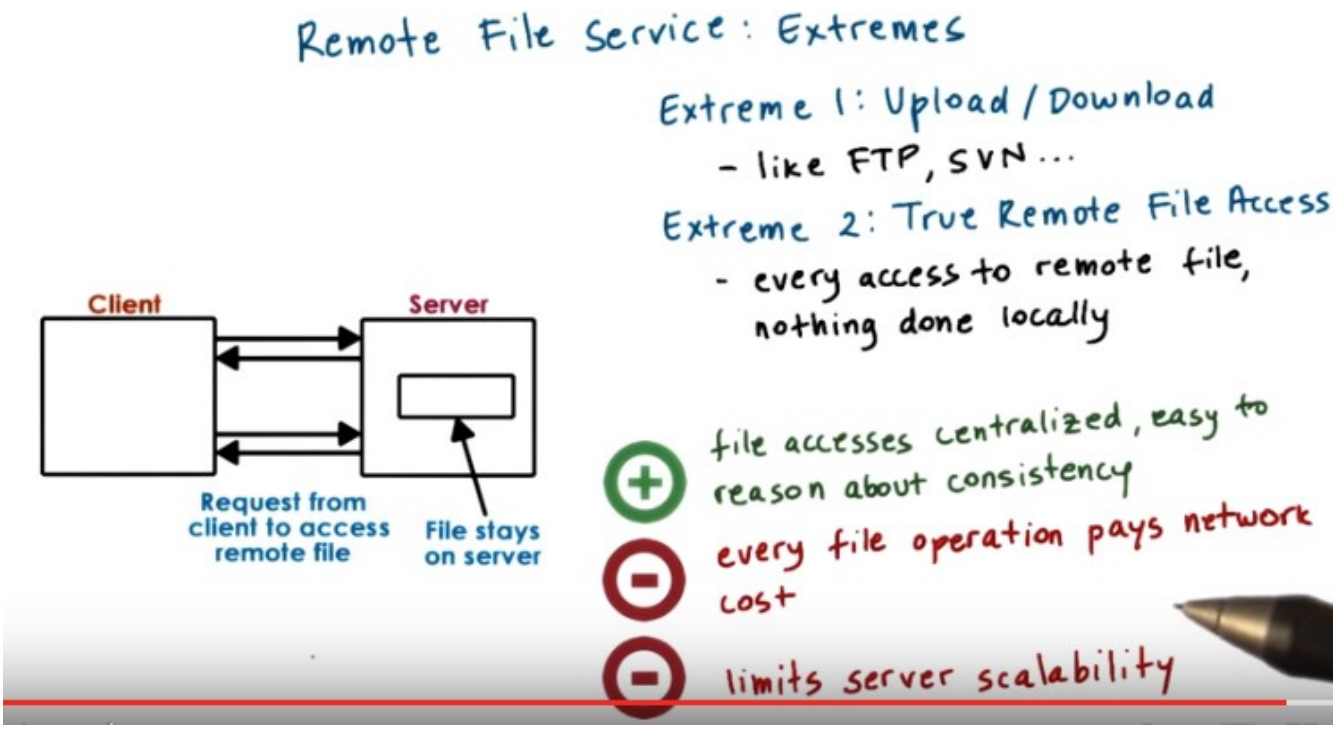
The second model is when the file server is not just single machine but instead, it's distributed on multiple machines. This may mean that all the files are replicated, and available, on every single machine. This helps in the event that there are failures, because there's always a replica of the file that's available on one of the other servers. And this also helps with availability since all the requests that are

coming in can be split across these different servers. Another way that the file server can be distributed among multiple machines is by splitting the files, dividing them, or partitioning them. So that different physical machines store different files. This makes the system more scalable than the replicated model because if you need to store more files, you simply add additional machines. In the replicated model, we still need every single server to store all files so if we need to scale the file system to be able to store more files, we'll have to buy machines with larger discs. At some point, unfortunately, will likely reach a limit, whereas with the partitioned model, we can just keep adding additional machines. Or we may use some combination of both approaches, replication and partitioning. For instance, we can partition all the files and then each of the partitions can independently be replicated across multiple machines. A lot of the big data file systems that are used by companies like Google and Facebook use this kind of approach. Finally, the files may be stored on and served from all of the machines in the system. This solution blurs the distinction between client machines and server machines. Because all the nodes in the system are peers in the sense that they're all responsible for maintaining the shared files and also providing the file systems service. Each peer in the system will take some portion of the load by servicing some requests, likely those that are for files that are locally stored on the peer machine. In this lesson, we will primarily focus on discussion of the first model, clients and separate server machines. And, we will focus on issues related to caching and consistency management in such environments. Note that, in the next lesson, we will discuss, more generally, issues related to distributed state management, when there are multiple peer nodes in a distributed environment, that are involved jointly in the management of some distributed state. Then we will talk about memory ascending sample but there will be a lot of similarities between the mechanisms that will describe for distributed memory management and the ones that one could apply for distributed file management in this kind of model.

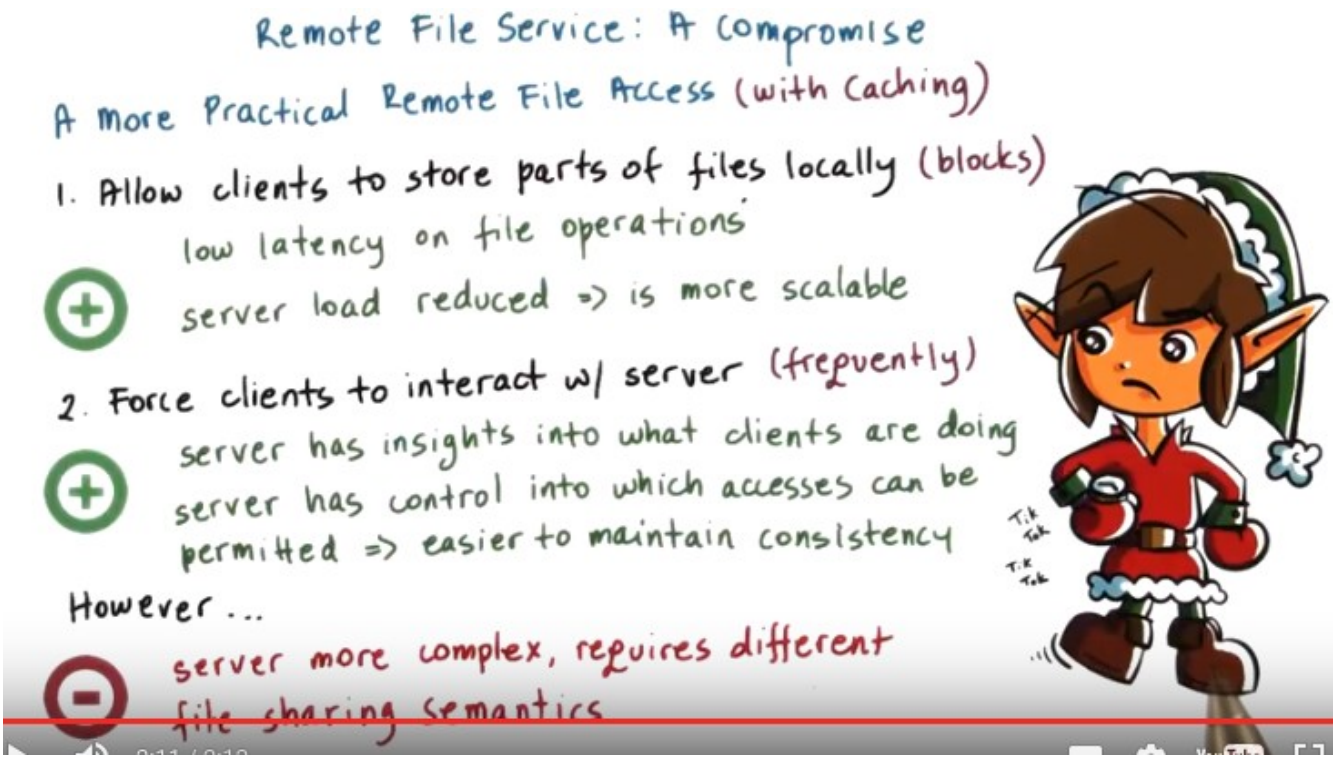


5. Let's look at the different ways that a remote file service can be provided. We will assume that there is one client and one server. Whereas that one extreme we have what it is called the Upload/Download model. This means that when the client wants to access the file, it downloads the entire file, performs locally the operations and then when done, it uploads the file back to the server. This is less similar to proper file system but it's more similar to the way FTP servers behave or even the way version management systems like SVN servers behave. The benefit of this model is that once the client has the file, it can perform all of the operations locally so that can be done fast. One downside of this model is

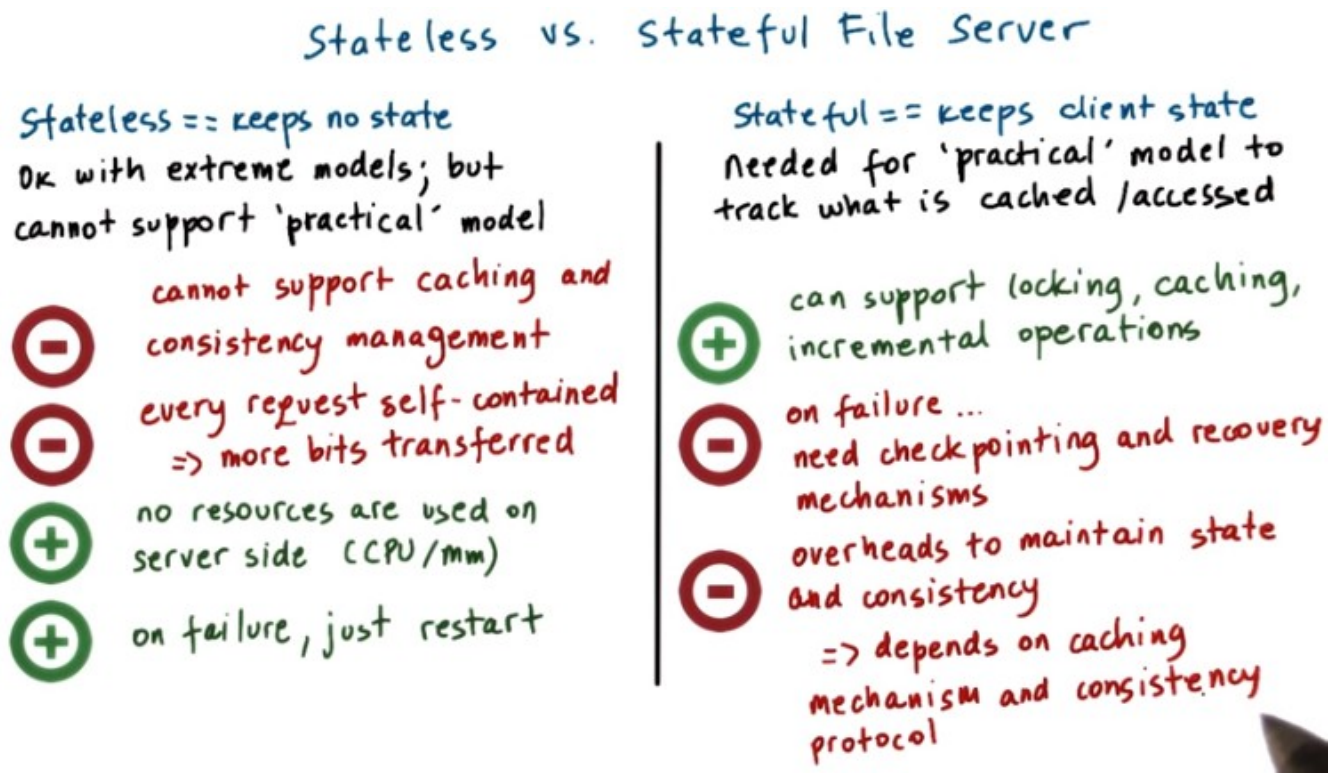
that the client needs to download the entire file even for a small file access for a small file modification. A second downside with this model is that it takes away file access control from the server. Once the server gives the file to the client, it has no idea what the client is doing with it, or when will it get it back. This makes some operations related to sharing files a little bit difficult to execute.



At the other extreme, we have a model that's called the true remote file access. In this model, the file remains on the server and every single operation, every read or every write, has to go to the server. In this model, the client makes no attempt whatsoever to leverage any kind of local buffer cache, local disks, or anything like that. Everything has to go to the server. The **benefits** of this extreme is that the server has full control and knowledge of how the clients are accessing and modifying the shared state and the shared files. This makes it easier to ensure that the state of the file system is consistent, and that there are no situations where multiple clients are overwriting the same portions of the single file at the same time. The **down side** of this model is that every single file operation has to incur the cost of remote network latency. In this model, the latency costs are suffered even when clients are only reading repeatedly from a read-only file. This model will bypass any resources that the client has locally, to cache that file. Also, since every single file operation goes to the server, the server will get overloaded more quickly, meaning that the single server will not be able to support as many clients. This limits the scalability of the file system solution.

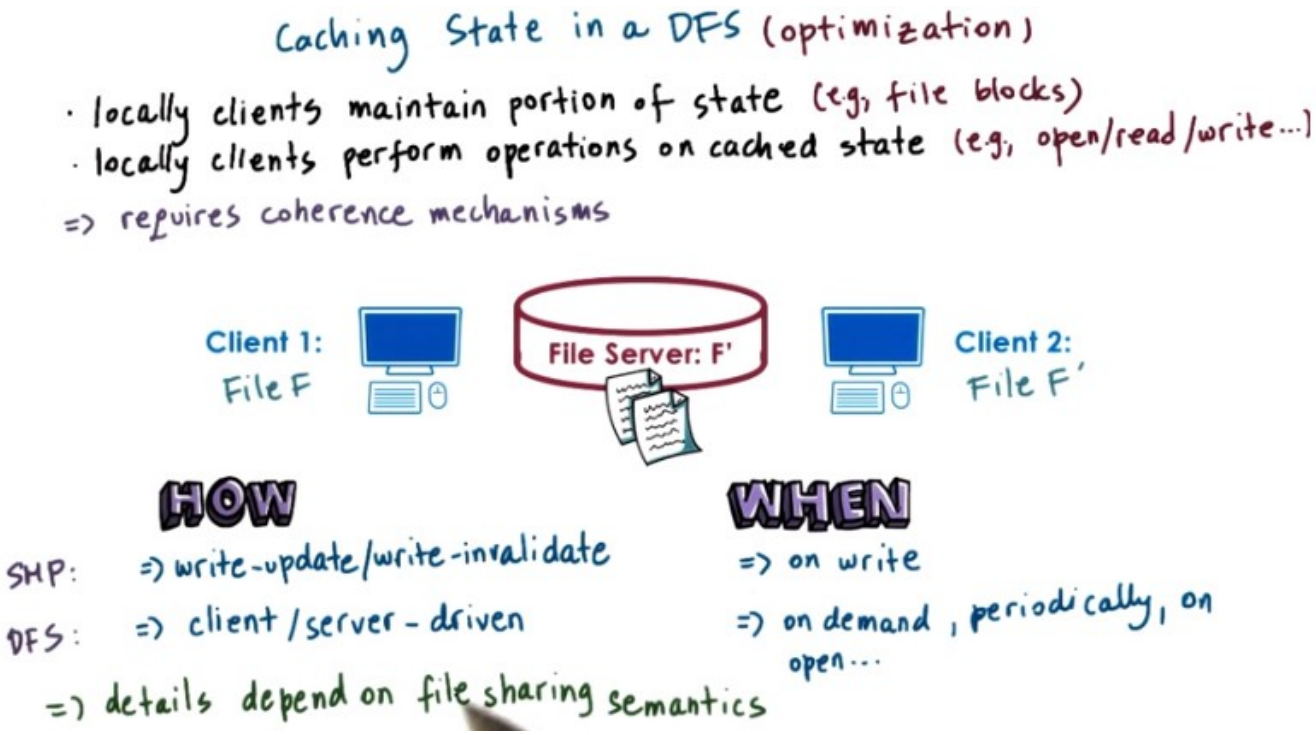


6. The two models are two extremes, but in reality, what makes sense is to have something in between. First, we should allow clients to benefit from using their local memories and their local disk, and to store at least some parts of the file they're working on. For instance, it makes sense for clients to download some blocks of the file that they're accessing. Or even to apply certain prefetching techniques just like what regular file systems do when they're trying to prefetch blocks from disks into their memory cache. This will lead both to lower latencies for those file operations that acts as these locally stored or cached portions of the file. And by doing this, also some load is removed from the server since some operations can be fully performed on the client machines. And thus the server becomes more scalable. Now, once we allow the clients to store portions of this file locally and to perform operations on the file, including modifications, it becomes necessary for the clients to start interacting with the server for two reasons. First, the clients need to notify the server of any modifications to the file that they have made. And also, the clients need to find out from the server if any of the files that they have locally cached has been modified by someone else. These interactions have to happen with some reasonable frequency so that it's not too late by the time that we discover that certain changes have been made. This is beneficial because unlike in the upload-download model, with this, the server is still in the loop. It has insights into what exactly are the clients doing, and it has some control over which accesses should be permitted versus not. So it becomes easier to maintain consistency. However, the problem with this is that it makes the file server more complex. It means that the server would have to perform additional tasks and maintain additional state so as to make sure that it can provide consistency guarantees. And achieving a practical solution of this model means that the clients would have to understand somewhat different file sharing semantics compared to what they are used to in a typical local file system.



7. Before we can discuss some of the design and implementation challenges with the models of remote services that we described in the previous videos, we need to distinguish among stateless and stateful file servers. First, the **stateless server** is one which doesn't maintain any information regarding which clients access which files, how many different clients are serviced, nothing. Every request has to be self-described, self-contained (think of the request header), so that it has all of the parameters regarding the filings being accessed, the absolute offset within that file, along with any data that needs to be written. This model is suitable for the upload-download model, or the other extreme, true remote file access service. **Downside**: But, it cannot be used for the more practical model, which relies on caching, because without state, we cannot achieve consistency management in the distributed file system. This is one of the biggest negatives of this approach. It prevents the use of caching, which is an important performance optimization technique. The other downside is, because all of the requests have to be self-contained, there will be more bits that will need to be transferred in order to describe each request. There are **benefits** to the approach as well. For instance, since there is no state that's maintained on the file server, no resources will be consumed on the server side to maintain that state. No memory will be used, no CPU will be spent, in order to make sure that the state is consistent, so that's one benefit. The most important benefit of the approach is that, because it is stateless, and because the requests are self-contained, this leads to a very resilient(有彈力的,愉快的) design in terms of failures. If the server crashes, it just needs to be restarted, and it can continue to service all of the future client requests without any potential errors. Sure, the clients will need to reissue any of the requests that have timed out, but they will ultimately receive the exact same type of answers, regardless of the fact that the server has failed. Neither the clients, nor the server, will need to do anything else special in order to recover from this failure, just restart. In contrast to this, a **stateful server** is one that maintains information about the clients in the system, what are the files they're accessing, what types of accesses they're performing, reads versus writes. For instance, for each file, the file system may track information, who has portions of the file cached, who has read or written to it recently, or similar information. **Benefits**: Because of this state, it becomes possible for the file server to allow data to be

cached and at the same time to guarantee consistency. Something that wasn't possible with the stateless server. And also to provide other benefits, like locking. We need state to make sure that we can keep track of who has locked the file, and whether or not the file is locked. Incremental operations is another thing that can be supported by having some state. A client can issue a request, I want to read the next kilobyte of data from this file. With a stateless design, there is no concept of what the next kilobyte would be. You have to describe every single request fully, with the offset of the file, with the specific file handle, and everything. The **problem** with this is that on failure, all that state that the server maintains needs to be recovered, so that the file system overall is still in a consistent state. That requires some more complex mechanisms, for instance, the state needs to be checkpointed. We have to have to some way to rebuild it in order to have a good representation of what it is that the clients were doing before the failure occurred. And of course, the runtime overheads with the stateful server because it needs to maintain this state, and also execute all of the necessary consistency protocols. Exactly what those overheads will be, will depend on the details of how caching is enabled, when and how it can occur, and then what are the consistency requirements that the distributed file system needs to support. We will look at what are some of the options in this space in the next videos.

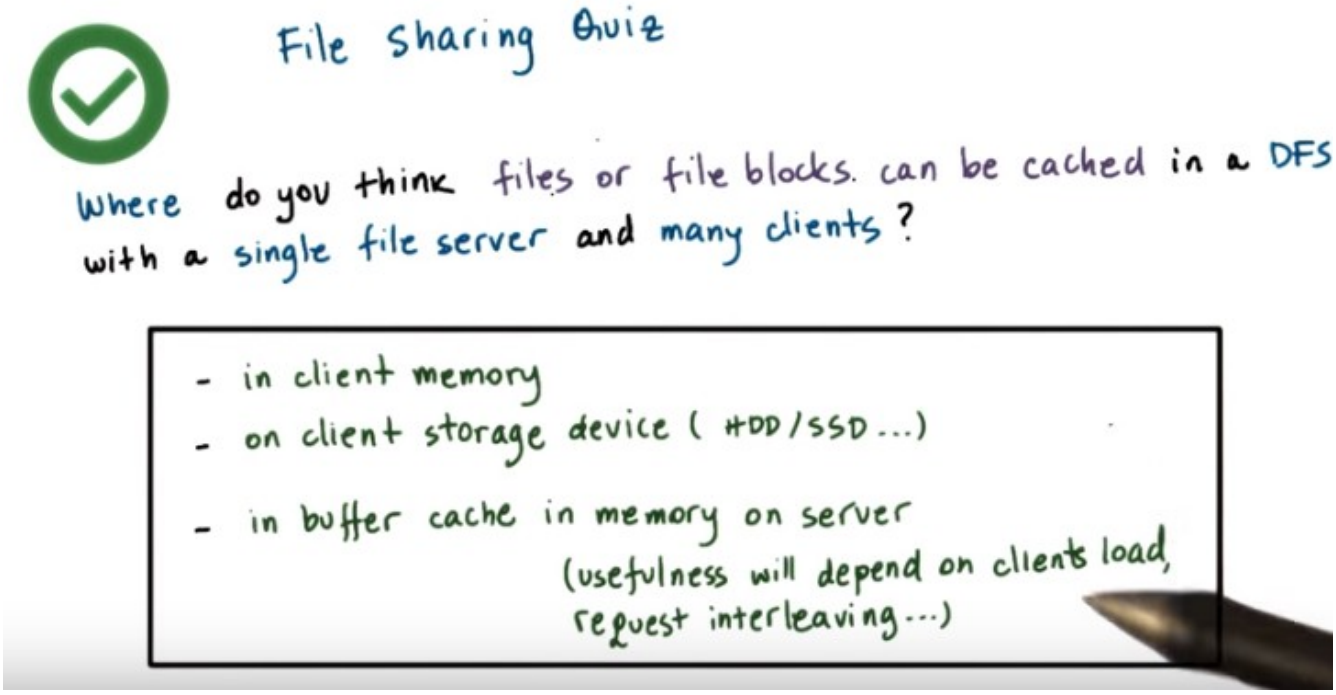


SMP: shared memory multi-processors
DFS: distributed file system

8. The first distributed file system mechanism we will look at is caching. Caching is a general optimization techniques in distributed systems, where the clients are permitted to locally maintain a portion of the state, in this case, portions of the files or file blocks. And also the clients are permitted to perform some operations on that cached state locally. For instance, some of the open or read or write operations can be issued against the locally cached files or file portions. This can be done potentially without contacting, and overloading the actual file servers. Keeping the cached portions of the file consistent with the on server representation of that file, requires that we have some coherence mechanisms. For instance, in this illustration, clients one and two both cache a portion of the file F.

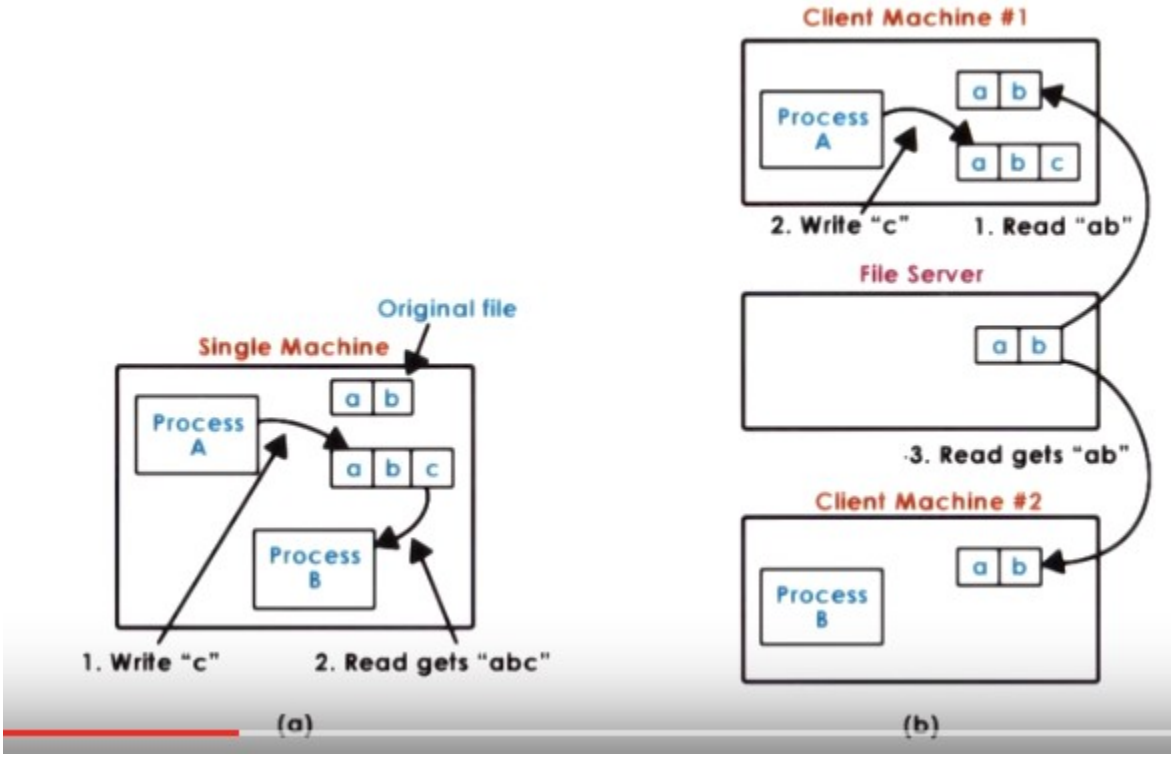
However, Client 2 has subsequently modified that file, F prime, and has also updated the file server to reflect those changes. The question then is, how and when will Client 1 find out that Client 2 has performed these changes? The problem here is similar to maintaining cache coherence in shared memory multi processors. There we said that we use mechanisms like write-update and write-invalidate, and these mechanisms get triggered whenever a particular variable, or a particular memory location gets written to. What this would mean in the context of this example that whenever Client 2 performs any kind of update to some portion of file F in its cache, that, that would be propagated to Client 1. Either as a write-invalidation message or a write-update, the actual change will be visible here. But given the very different communication costs, and also latencies that exist in distributed systems, achieving this may not be realistic. And also, it may not even be necessary, given the ways that files are being shared. Instead for distributed file systems, things that make sense would be to trigger some of these coherence mechanisms on demand when the client needs to access a file or periodically whenever the client is open. And when exactly get executed will also depend on whether the coherence mechanism is something that is client driven so the client initiates. I need to find out if the file is updated. I need to see the new version of this file. Or, server-driven, where the server notifies the clients who have cached the file, in this case, Client 1, that something has changed about their cached state. The exact details of how and when the coherence mechanisms are executed have to be such so that the file system can maintain consistence guarantees. However, what those details will be will depend on the file sharing semantics that this distributed file system needs to support.

9. Before we move on, I would like to ask you a question to collect your initial thoughts about file caching in a Distributed File System. Where do you think files, or file blocks can be cached in a Distributed File System that has a single file server, and many clients? Type your answer in the text box, and then compare it to my solution in the following video.

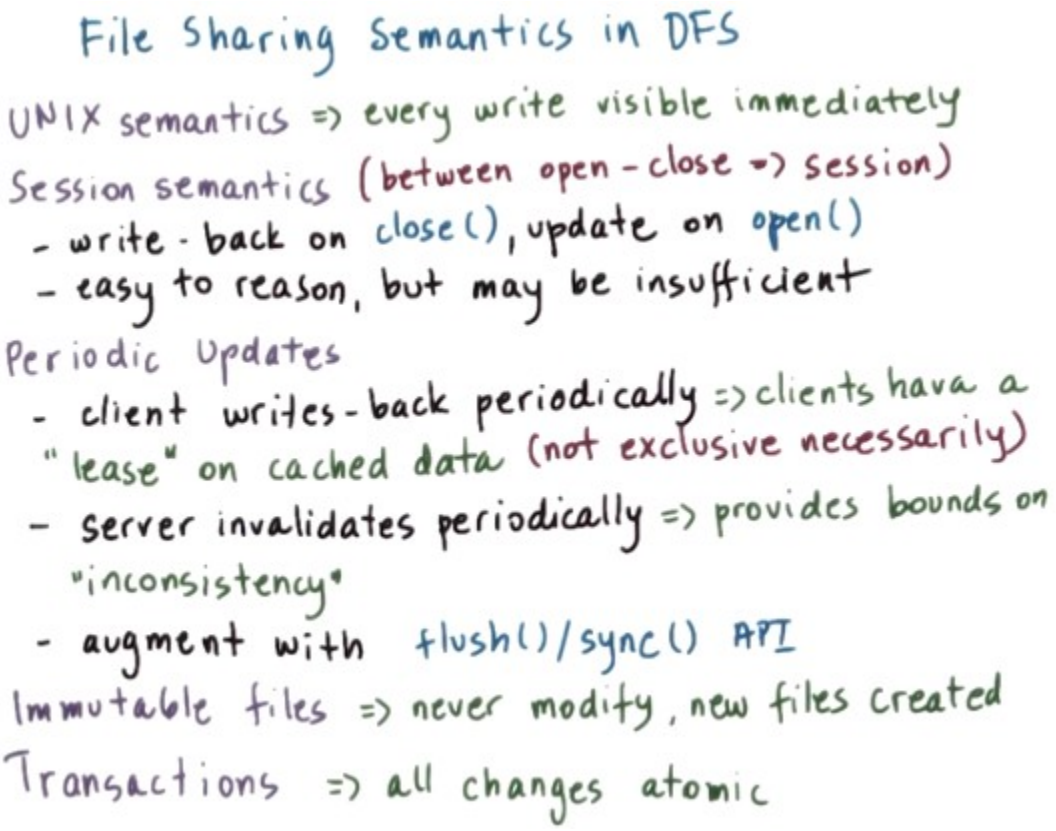


10. First, the files or the file blocks can be cached in the client memories in their buffer cache. The first place where files or file blocks can be cached will be the client's memory. As the files or the file blocks are brought in from the server to the clients, they can be present in the client's memory as part of their buffer cache. This is what regular file systems do when they're bringing in files they're reading from

their local discs. Second, the clients may store cache components on their local storage devices, hard discs, SSDs. It may be faster to retrieve portions of the file from the local storage than to go via the network to the remote file system. Finally, the file blocks can also be cached on the server-side in the buffer cache in memory on the file server machine. However, how useful this will be will be the hit rate on that buffer cache will depend a lot on how many clients are accessing the server at the same time, and how are their requests interleave. If there is high request interleaving, the buffer cache may prove not to be very useful. Because there may not be much locality among the accesses which are originating, in this case, from many, many clients.



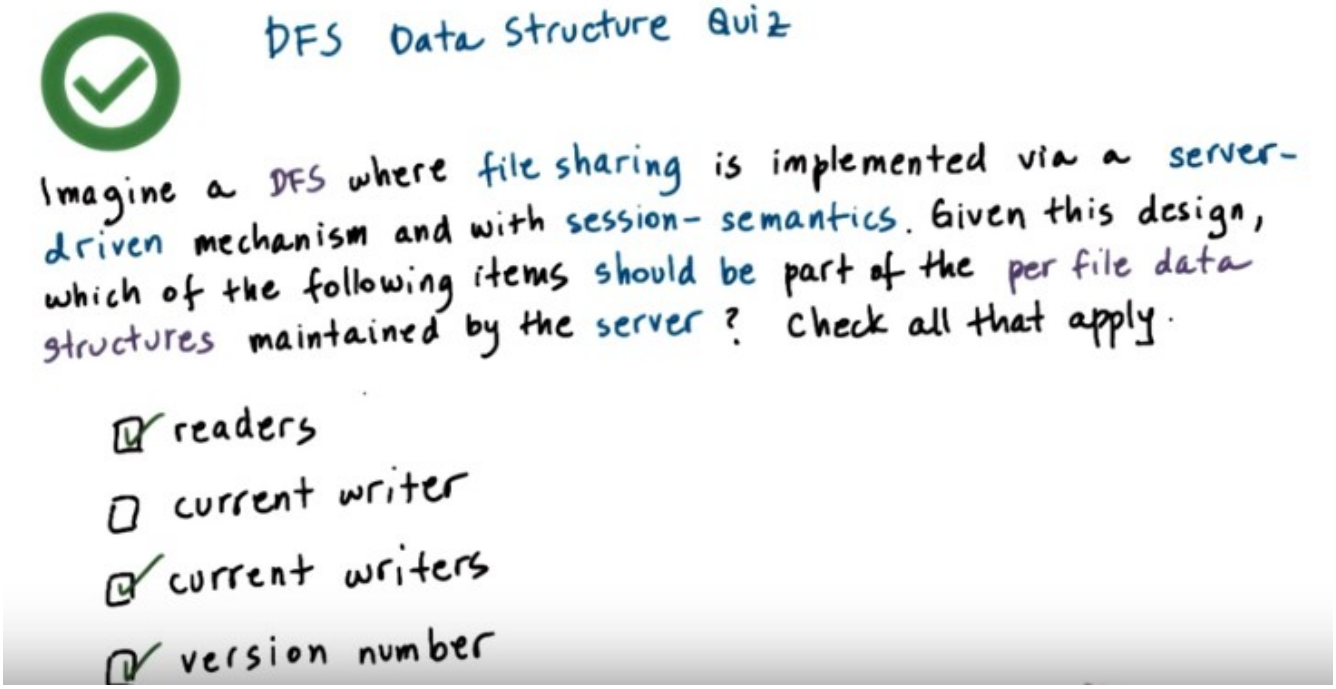
11. To explain the file sharing semantics in a distributed file system, let's see what these look like first in a single node environment (圖 a) and then compare how they differ from the distributed multi-node environment (圖 b). On the **single node** in a Unix environment, we have the following semantics. Whenever a file is modified by any process, in this case process A write c, that change will immediately be visible by any other process on that machine. So Process B will immediately see that c when it performs a read on this file, will get abc. This will be the case, even if the change isn't actually pushed out to disk, because both of these processes have access to the same buffer cache. In **distributed file systems**, that's not the case. Even if the fact that Process A performed an update, wrote c to this file, gets pushed to the file server immediately, that message, it may take some time before that message actually arrives here. So it is possible for Process B to not be able to see that message for a while, and whenever it performs a read operation on that file, even in that period after this file was written to, it will continue seeing that the file consists of only the elements a and b. Given that message latencies may vary, we have no way of determining how much should we delay every possible read operation in order to make sure that any write that may have happened anywhere in the system arrives to the file server, so that we can guarantee that this Process B does read the most recent version of the file. So in order to maintain acceptable performance, distributed systems would typically sacrifice some of the consistency, and they will accept some more relaxed file sharing semantics.



Let's see what are some possible, meaningful file sharing semantics for distributed file systems. As a baseline we will contrast this to the case when your file system is on the single machine where every write is immediately visible, and this is what's referred to as **UNIX semantics**. Something that makes sense to do is to enforce what we call **session semantics**. Whenever a file is closed, the client writes back to the server all of the changes that it has applied to that file in its cache. Whenever a client needs to open a file, it doesn't use the cache contents, instead, goes and checks with the file server whether or not there is a more recent version of that file. We call the session semantics with the period between the file open and the file close being referred to as one session. With session semantics it is very possible for one client to be reading a stale(陳舊的) version of a file while another client is either updating that file or even reading a more recent version of the file. But at least by knowing that when we open or when we close the file, we will be consistent with the rest of the file system at that particular moment. We can reason something about what kind of sharing guarantees do we have when we run in this type of distributed file system? Although session semantics are intuitive to reason about, they're really not good for situations when clients want to concurrently share a file, write to it and see each other's updates, then have to open it and close it repeatedly. And also when files stay open or are being updated for long periods of time, these can result in long periods of inconsistency in the file system. **Periodic updates**: For that reason, it makes sense to introduce some time intervals when these updates happen. With introducing time intervals, the client updates, writes, will be propagated to the server periodically. One way to think about this is that the client has some sort of lease on how long they can use the cached data. However in this case know that we don't really mean that it's an exclusive lease, like locking. This is a completely separate issue. In the same way, the server notifications, the invalidations, are also periodically sent out to the clients. This can establish some sort of time bounds during which the system can potentially be inconsistent. So at least if there are any conflicts, it will be easier to correct for them. There will be likely fewer of them that have accumulated

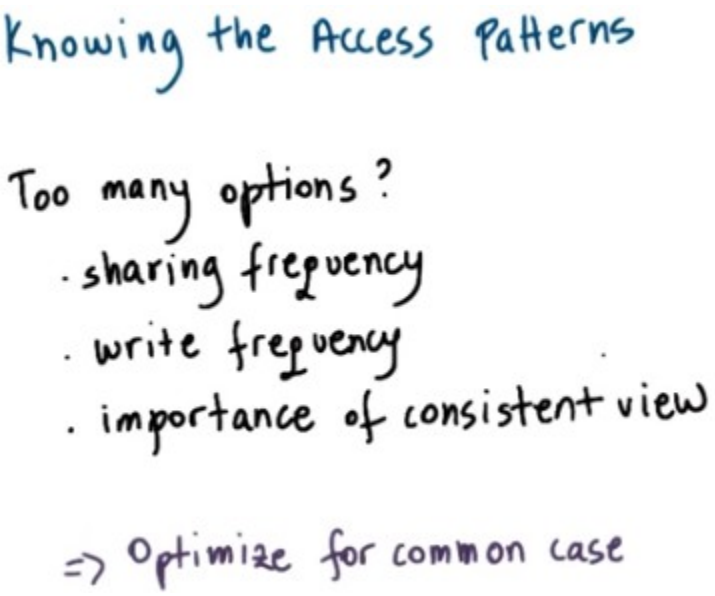
during the period of these updates. Since the client doesn't really have any idea about what are the start and the end times of these synchronization periods, the file system can also provide some explicit operations to let the client flush its updates to the remote server. Just like what we do with flushing the updates to disk when it comes to local storage, and also to explicitly sync its state with that of the remote server. Again these types of operations are not necessarily distributed file system specific, they're used in regular, local file systems as well. Other file sharing policies also exist and make sense for certain situations. For instance, files may be simply **immutable**, you never really modify a file, you simply delete it, or you create a new file with a new name. When you're sharing photos via Instagram or Facebook, you don't really upload a photo and then go and edit it, if you need to change the photo you change the photo and you upload the modified photo. These types of distributed storage have these kind of semantics immutable files. Another useful file sharing semantics would be for the file system to provide **transactional** guarantees. What this would mean is that the file system will need to export some interfaces, some API so that the clients can specify what is the collection of files or the collection of operations that need to be treated like a certain single transaction. And then, the file system can make some guarantees that all those changes are atomically committed, atomically made visible into the file system. As we work through the rest of the lesson, we will look at what are some of the file sharing semantics that are supported in some distributed file systems, and also what are the mechanisms that are required to achieve these types of semantics.

12. Let's take a quiz. For this quiz, I want you to imagine a distributed file system where the file sharing is implemented via a server-driven mechanism and with session-semantics. Given this design, which of the following items should be part of the per file data structures that are maintained by the server? The options are: readers, current writer, current writers, version number. You should check all that apply.

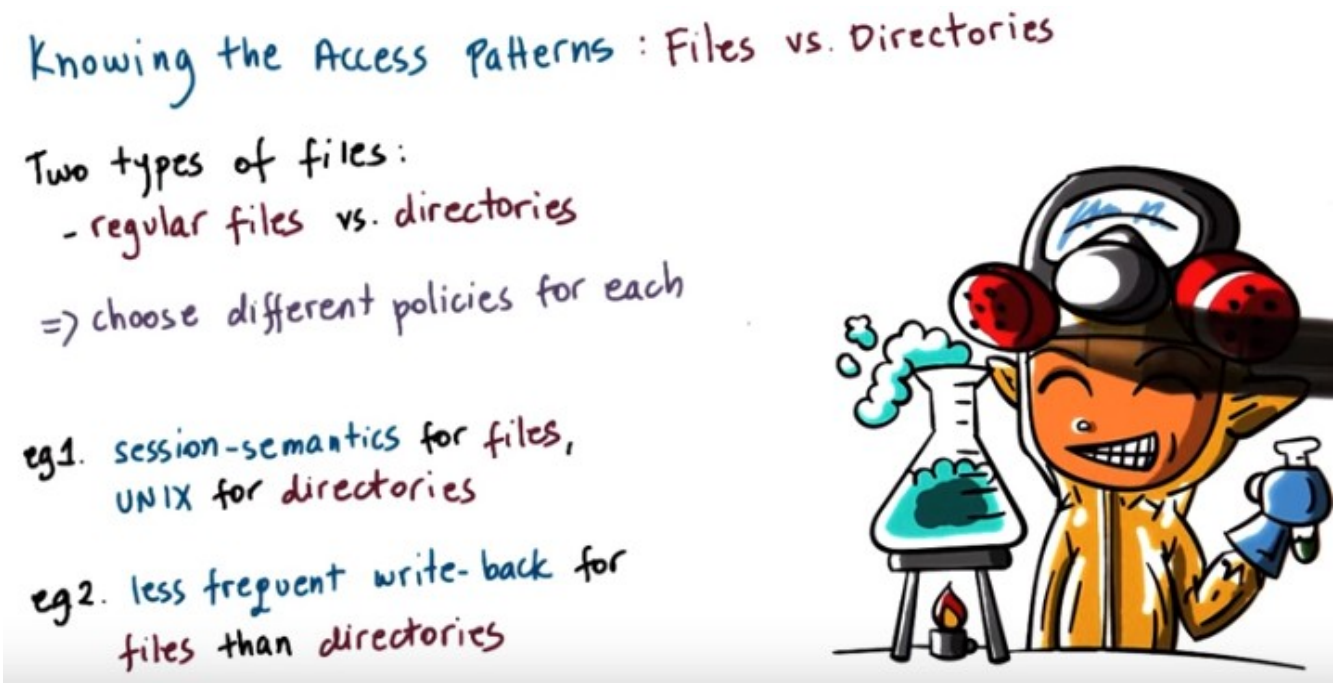


13. Before we answer the question, let me remind you, that a server driven mechanism means that it is the server that will push any invalidations to the clients. And also, that session-semantics means that any changes made to a file will become visible when the file is closed, when the session is closed, and when a subsequent client opens that file, starts a new session. So that means that it is possible for

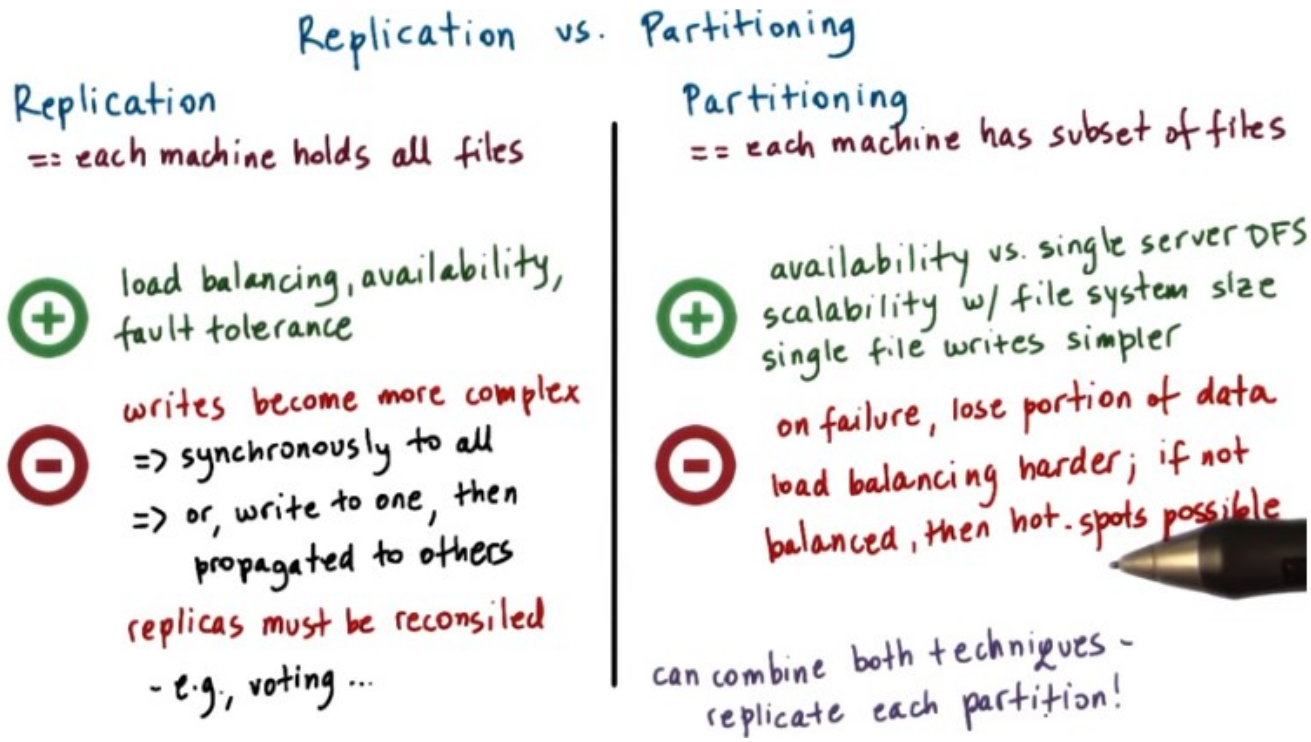
overlapping sessions to see different versions of the file, so it is possible to have concurrent writers. Session semantics doesn't specify what will happen to the file when all of these writers close that file. Whether one of the versions will become the valid one, or whether they will be merged in some ways, or whether some error will be raised back to the client so that they can resolve the conflicts. So given this information, the items that make sense to be part of a per file data structure in this kind of distributed file system will include information about what are the current readers of the system. Also, what are all of the current concurrent writers, potentially multiple, not just one. And also, it makes sense to keep track of something like a version number so that the clients know which version were they given, and the server also understands which clients have been modifying an old version of the file versus the newest one.



14. As we mentioned multiple times in this course, understanding the workload, in this case, that will be the access pattern, how the files are being accessed. This is an important piece of information, that's useful when we're trying to design and optimize a particular solution in a certain way. When we're thinking about how to design the system so as to support a particular file sharing semantics, we need to understand things like what is the sharing frequency, what is the write frequency, so what is the access pattern. And also how important is it to maintain a consistent view for that particular type of files. Once we understand these workload properties, the design of the file system must be done so that it's optimized for the common case.



One issue is, however, that file systems have two different types of files, regular files and directories. In these two types of files, we often have very different access patterns in terms of what is the locality, what is the lifetime of the files, the size distribution, how frequently are they accessed. For these reasons, it is not uncommon for these two types of files to be treated differently. For instance, we can adopt one type of semantics for the regular files and another type of semantics for the directories. Or, if we use periodic updates as a mechanism for both, then we may choose to use less frequent write-backs for the regular files versus for the directories. This can be based on observations that, for instance, directories are more frequently shared, than individual files in them. Later in this lesson, we will look at the choices, how to treat these two different types of files, for the network file system manifest, and the sprite file system.

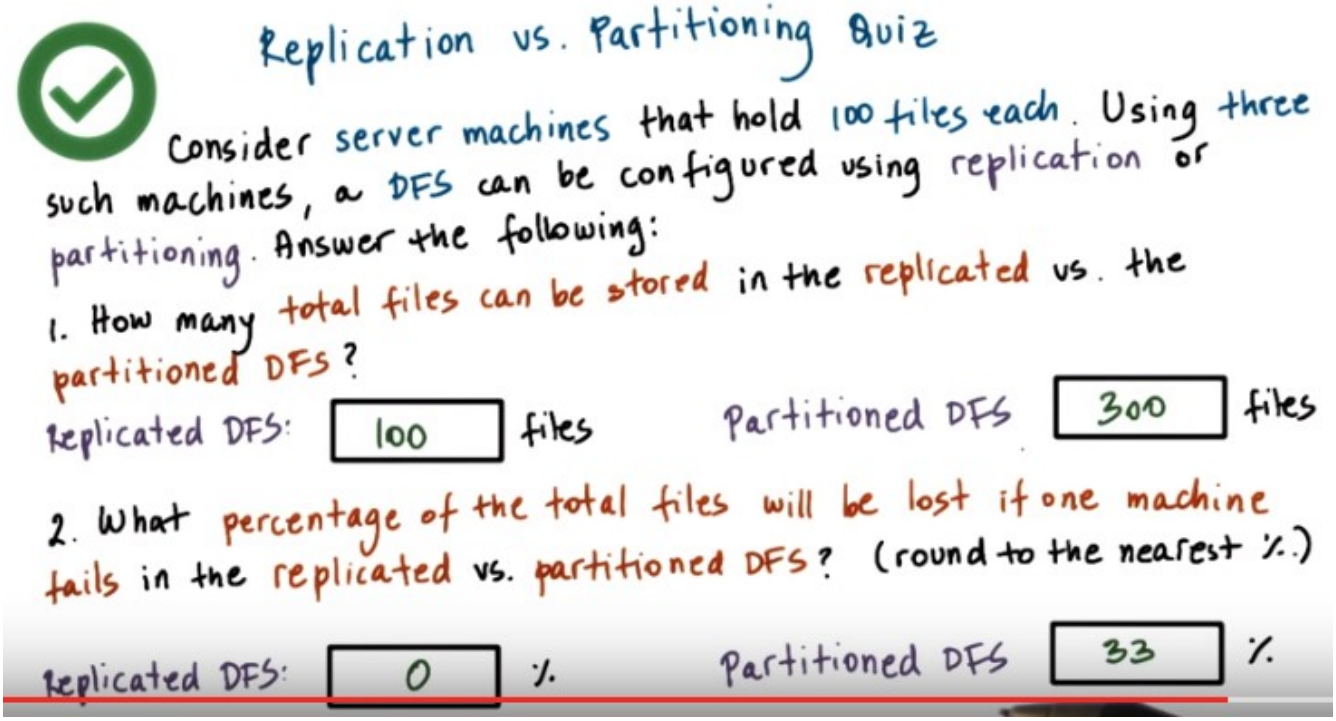


reconcile: 使和解,調停. (reconsile 查不到)
注意右邊是 subset of files, 不是 subset of file

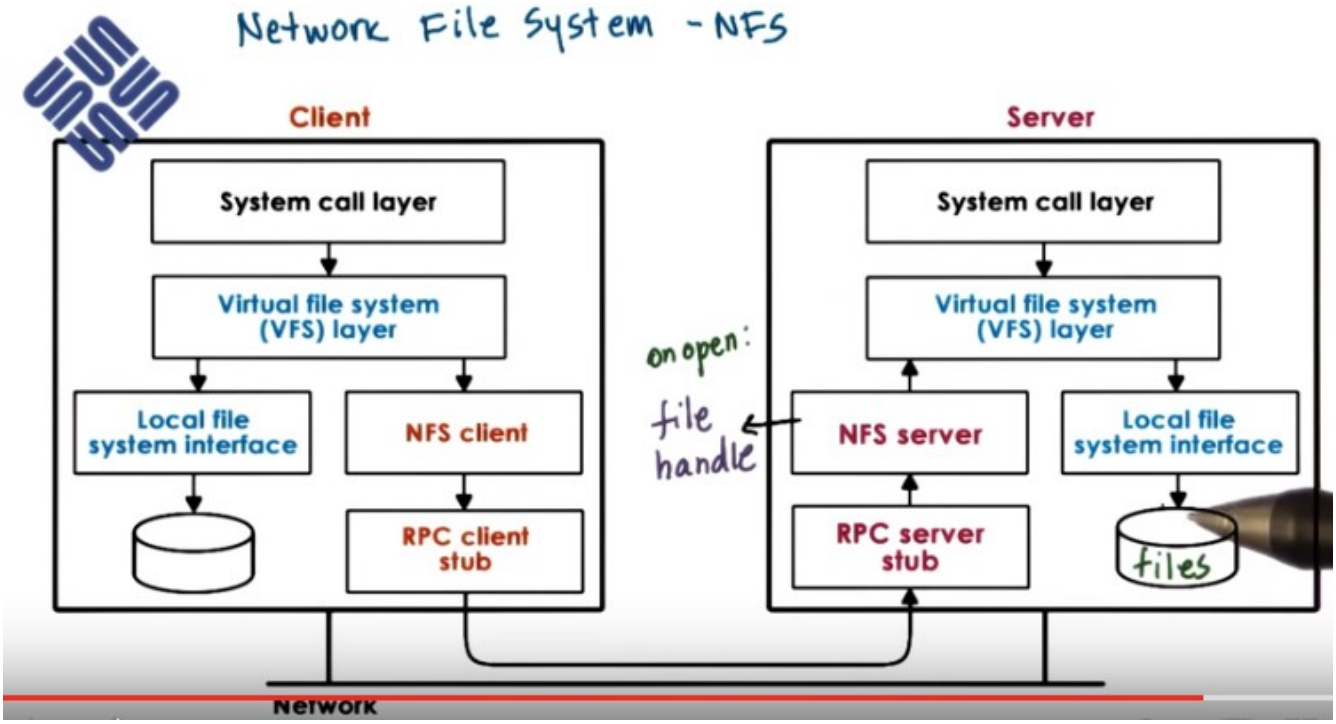
15. Before moving onto concrete examples, I want to explicitly mention one more design dimension when it comes to distributed file systems. We said that the clients and the server can be distributed to different machines, but the file server itself can also be distributed. This can be achieved via replication or partitioning. With **replication**, the file system can be replicated onto multiple machines, such that every single machine holds an exact replica of all of the files in the system. The **benefit** of this can be that the client request can be load balanced across all replicas, across all machines. And this can lead to better performance. The system overall can be more available, it will return responses more quickly. And also it is more fault tolerant. When one replica fails, the remaining replicas can continue serving the data for all the files. The **downside** is that now the write operations that update to the file system state, may become more complex. Not only do we have to worry about the consistency among the clients that may cache the file, and the servers, but also about the consistency among all of the replicas. A simple solution is to force every single write to each replica. And only after that is done to actually return to the client and to consider that the write operation has completed. This will slow down all writes. An alternative would be to allow the writes to be applied to one server, to a single replica copy. And then to have some background process that asynchronously propagates these writes to the other replicas. 此句跟上句無關：If it turns out that there are any differences among the state of a file on different replicas, then these differences need to be resolved. For instance, we can use a simple technique like voting, where the votes are taken from all servers and majority wins. There are many other techniques how these sorts of issues can be resolved, but these are beyond the scope of this course. The other technique is to distribute the file system state using **partitioning**. As the name suggests, in partitioning every single machine has only a portion of the state, only a subset of all the files in the system. This may be done based on file names. For instance, all the files from a to m sit on one machine, and all the files from n to z sit on another machine. Or we may choose a policy where different directories are stored on different machines, where we'd somehow partition the hierarchical

name space of the directory tree structure. There can be various criterias that can be used to decide how to partition all the state in the file system. **Benefits**: Using this technique, we can definitely achieve greater availability compared to a single server design. Each server here will hold fewer files and therefore, will be able to respond to a request for those files more quickly, so will appear to be much more available. The most important benefit of this design is that it provides for greater scalability when we consider the size of the file system, the overall size of all the files stored in that file system. With replication, given that every server has to hold all the files, the size of the file system will be limited by the capacity of a single machine. In partitioning, if we need the bigger file system, we just add more machines and problem solved. And finally, unlike in the replication case, in the partitioning case when we need to perform a write to a single file, that will remain localized to a single machine. So that's much simpler than what we have here. One of the main **problems** with this approach is that when there's a failure, a portion of the data in this file system will be lost. So, all of the files that are stored on that particular machine, the machine that's failed, will not be accessible anymore. In addition, balancing the system is more difficult because we have to take into consideration how the specific files are accessed. If there is a particular file that's more frequently accessed by most clients in the systems, then that will create hotspots. Finally, these two techniques can be **combined** to have a solution where the files are partitioned across different groups or in different volumes. And each of these groups is then replicated, potentially with different degree of replication. For instance, you can have partitions of read-only files versus files that are also written to, and you can replicate the read-only files to a greater degree. Or you can consider having smaller partitions where there are files that are more frequently accessed, versus larger partitions that consist of more files but less frequently accessed files. And then you can consider using different degrees of replication for the partition that has more frequently accessed files, versus less frequently accessed files. So that overall each machine has approximately the same number of expected client requests.

16. For a quiz, let's compare replication with partitioning. I want you to consider server machines that can hold 100 files each. Using three such machines, the Distributed File System can be configured using replication or partitioning. I want you to answer the following. First, how many total files can be stored in the replicated versus the partitioned Distributed File System? And second, what is the percentage of the total files that will be lost if one machine fails in the replicated versus the partition DFS. You should round your response to the nearest percentile.



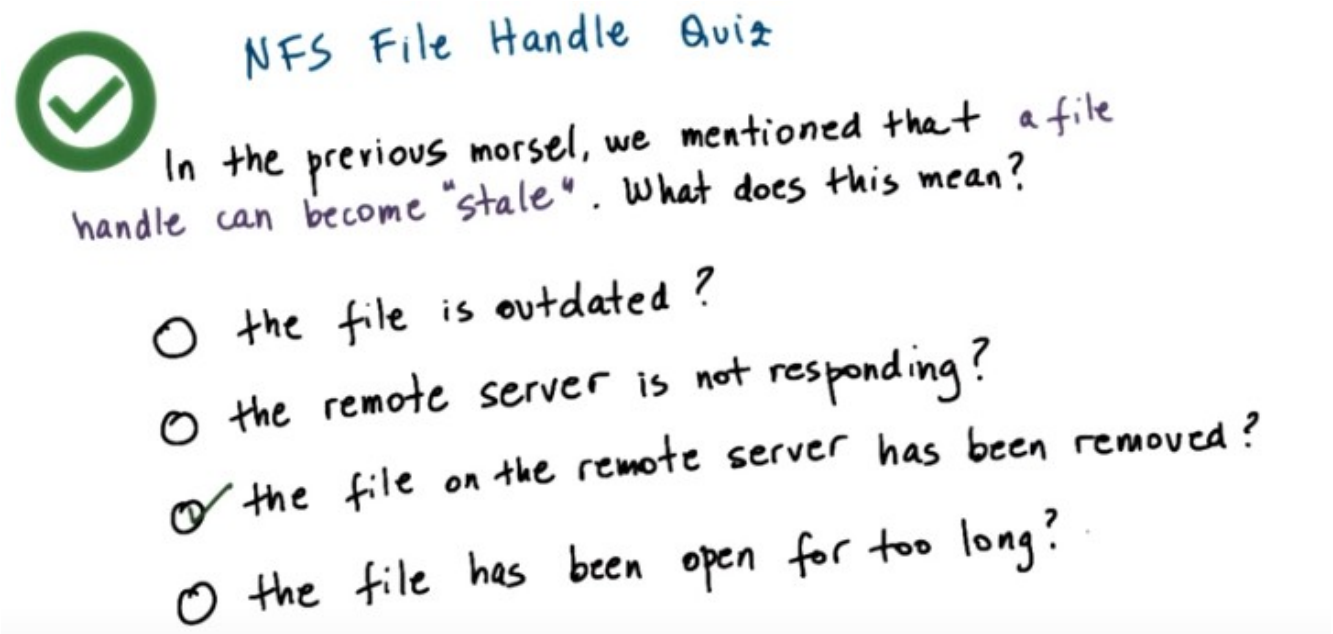
17. The answers are pretty straightforward. For the first question, we know each machine can hold 100 files in the replicated case, all files are present on every single machine, so regardless of the fact that we have three machines, we still can only hold 100 files. In the partitioned case however, every single machine can hold 100 different files, and therefore the total size of the file system is 300. This illustrates one benefit of the partitioned design, because it can support larger number of files in the file system. For the second question, if we lose one machine in the replicated case, we will still have two other machines that have all the files, so we will lose zero percent of the total files. In the partitioned file system however, 33% of the files will be lost. This is because when one machine fails, all the files that were stored on that particular machine will be lost. This data point illustrates one case in which the replicated design is better. It provides greater fault tolerance versus in the previous example, the partitioned distributed file system providing greater scalability in terms of the number of files. Also this is why a mixed approach of using both replication and partitioning types of techniques will provide greater flexibility in terms of achieving both size and resiliency.



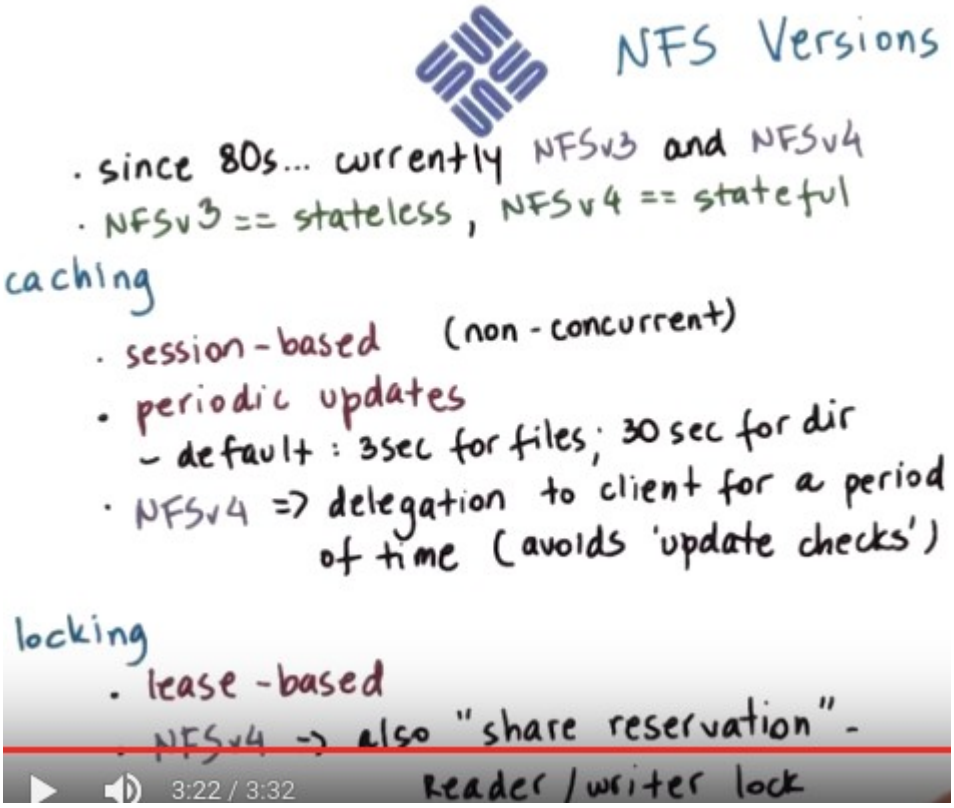
上圖底部的 RPC 應該就是 remote procedure call. 注意上圖中箭頭走向.

18. We will now look at NFS, a very popular distributive file system. In NFS, clients acts as the remote server over a network. And hence, the name Network File System. It's another contribution to computer science made by the great systems researchers at Sun. In fact, one of the reasons why protocols like RPC were developed was to help with the use of NFS. Its architecture is shown in this figure. Client requests an access files using the virtual file system (VFS) interface and using the same types of file descriptors that they use to access files in their local storage. The VFS layer will determine whether the file belongs to the local file system or whether it needs to be pushed to the NFS client, so that it can pass it to the remote file system. The NFS client interacts via RPC with they NFS server that resides on a remote machine. This is the machine that actually stores the files. The NFS server accepts the request, forms them into a proper file system operation that's then issued to the local virtual file system and from there, it gets passed to the local file system on top of the local storage. What that means is that on the server machine, the requests that are coming from the NFS server module are serviced as any other file system operation that comes from any application running on this machine. When an open request comes from the client, the NFS server will create a file handle. This will be a bite sequence that encodes both the server machine as well as the server local file information. This will be returned back to the client machine and it will be maintained by the NFS client, so whenever the client application tries to access files that are stored on the remote server on the NFS. Internally, this file handle will be passed with every single request. If the files get deleted or the server machine dies, using this handle will result in an error, because we're trying to use stale(陳舊的) data. Data that's no longer valid. On client's write operations, the data that needs to be written to the file will be carried as part of the RPC request from the client to the server machine. And in file read, the data blocks that will be read from the file will be the results from that RPC request that was sent from the client to the server. And as such, they will be passed back to the NFS client. And then ultimately, back to the application that issued the read operation.

19. Let's recap the design of NFS by asking a question about file handles. In the previous morsel, we mentioned that a file handle can become stale. What does this mean? The file is outdated? The remote server is not responding? The file on the remote server has been removed? Or the file has been opened for too long?



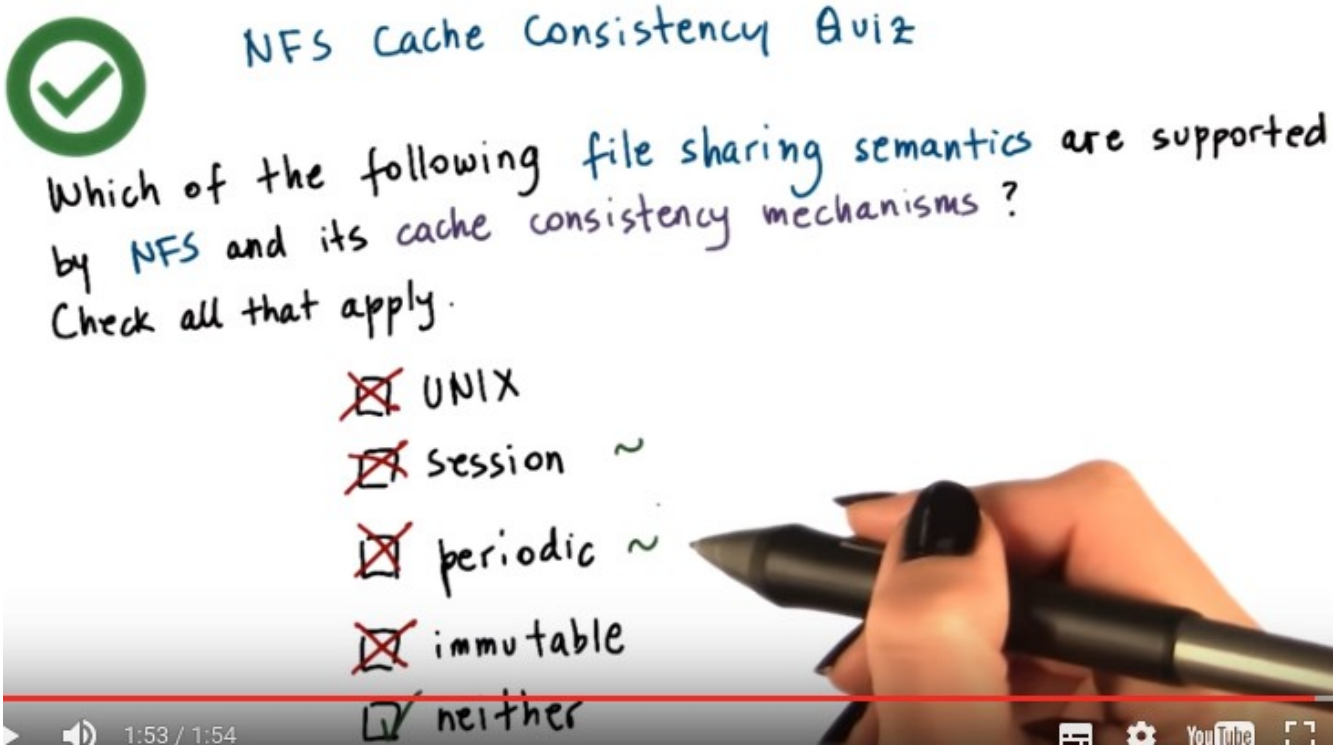
20. >From these options, the only correct option is the file on the remote server has been removed. (The first option:) The file is outdated really implies that the file has been written by somebody else potentially, since a particular client acquired the file handle. So that may be a consistency related problem, but it really doesn't return a stale handle. (The second option:) The fact that the remote server is not responding, that's really more of an RPC layer error. If the server is not responding that has nothing to do with our ability to access a file at some point or another on that machine. It may simply be a network error. And the final option is also wrong, the file handle will not become stale just because a file was open for too long. There may be some distributed file systems which provide only time places where a client is only allowed to keep a file open for a period of time, but NFS is not one of them. So for that reason this answer is not correct.



21. NFS has been around since the 80s and has gone through several revisions. The popular NFS versions that are in use today and that come standard with Linux distributions are NFS version 3 and version 4. There is one fundamental difference between these two versions, and that is that, according to the protocol specifications, NFS version 3 is stateless whereas NFS version 4 is stateful. The fact that NFS version 4 is stateful, allows it by design, to support operations like client caching and file logging. And although NFS version 3 is stateless, actual implementation of this protocol typically incorporate additional modules so that file caching and logging can be supported. The **caching** semantics in NFS are as follows. For files that are not accessed concurrently, NFS behaves with session semantics. On close, the changes are flushed to disk. For files that are not accessed concurrently, NFS behaves with session semantics. On close, all of the changes made to a file are flushed to the remote server. And then on open, a check is performed, and if necessary, the cached parts of the file are actually updated, so the new versions of those files are brought in. However, as an additional optimization, NFS also supports periodic updates. These are used to make sure that there aren't any changes on the file that the client is working with. And using these periodic updates, we'll break the session semantics, when there are multiple clients that are concurrently updating a file. The periods on when these checks happen can be configured but, by default, NFS uses these values: it uses 3 second checks for the regular files and, at 30 second intervals, it checks whether there are any changes with the directories. The rationale behind these numbers is that, the directories as files are modified less frequently, and that when modified it is easier to merge those changes. So we don't have to check for changes in the directories as frequently as we have to check for changes in the regular files, and still have a consistent system. NFS version 4 further incorporates a delegation mechanism where the server delegates to the client, all rights to manage a file for a period of time, and this will avoid any of the update checks that we described here. With server side state, NFS can support

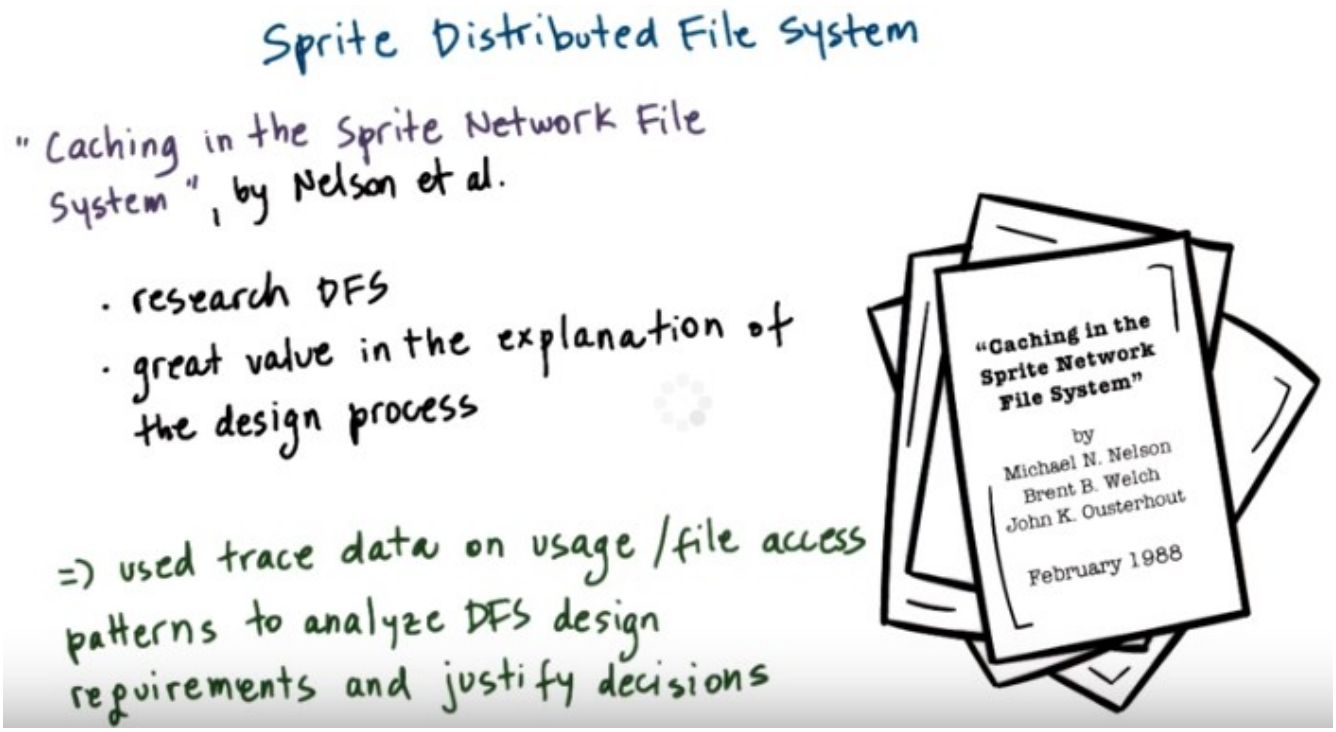
locking. The way NFS supports locking is using a lease-based mechanism. When a client acquires a lock, the server assigns it a particular time period during which the lock is valid. It is then the client's responsibility to make sure that it either releases the lock within the leased amount of time or that it explicitly extends the lock duration. This helps deal with situations of client failure, so in this case, when a client fails, on the server side, we'll just realize that okay, the lease for this client expired, so we can assign the lock to somebody else. And then when the client comes back again, or when network connectivity is re-established, the client will know that the lease expired, it no longer has a valid lock, so whatever changes it was trying to make, it simply has to redo. They weren't applied in a exclusive manner. NFS version 4 also supports more sophisticated mechanisms than just a basic lock. For instance, it supports a reader writer lock called share reservation, along with mechanisms serve on how one can upgrade from being a reader to being a writer for a file, and vice versa.

22. To recap again, the file sharing semantics supported by NFS, here is a quiz asking you how NFS maintains the cache consistency. Which of the following file sharing semantics are supported by NFS in its cache consistency mechanisms? Check all that apply from these options. UNIX, session, periodic, immutable, or neither.

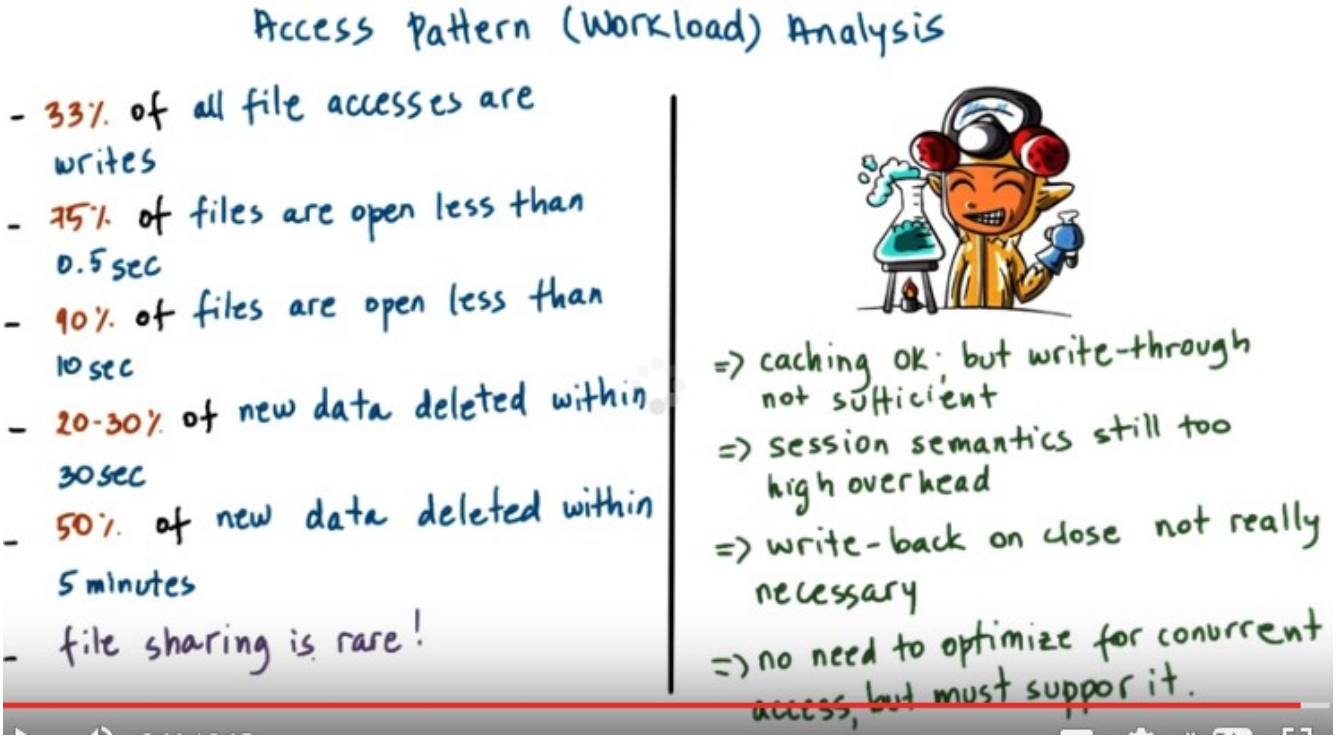


23. Let's see first which options we can eliminate quickly. NFS definitely allows for files to be modified, so immutable is not one of the correct answers. And also being a distributed system and if this doesn't meet guarantees that an update for a file will immediately be visible, so it's not Unix. Now we said that in principle, NFS tries to make session semantics in that the updates made to a file will be flushed back to the server when the file is closed. And also when a client performs an open operation, the client can check with the server to see whether the file has been updated. And in that case, the client will also update the cached value. The problem is however, that NFS can be configured to periodically have client and server interactions that check for any intermediate updates to our file during a session. Now how frequently this is done can be configured and in fact, it can be completely disabled. So, in that case, NFS will always behave like with session semantics. However, given that

this option for periodic updates exists. It is not quite a session semantics. At the same time, it's not purely periodic file sharing semantics, because we will still have changes in the file propagate and file close. Or on file open, just as what happens with a session-based file sharing semantics. So for both session and periodic, yes, perhaps there are elements of the sharing semantics that NFS supports that are session like or periodic like. And whether it will behave like with session or periodic semantics, it will really depend on how NFS is configured. That leaves that by default NFS is really neither. It is not purely session-based file sharing semantics distributed file system. And also, it doesn't purely support just periodic file sharing semantics.



24. Let's now look at another distributed file system example, the Sprite Distributed File System. And also the decision it makes regarding its consistency protocols. We will base our discussion on the Sprite Distributed File system as described in the research paper Caching in the Sprite Network File System by Nelson and others. This is an older paper, and it doesn't describe a production file system like when we talked about NFS, instead, Sprite was a research-distributed file system. But at the same time, it was also actually built and deployed at UC Berkeley, and people were using it. What's really nice about this paper is that it has a lot of detailed explanations of what was the usage pattern, the file access pattern, that motivated the particular design that Sprite has. The authors used trace data that was gathered from users using a real distributed filesystem to understand something about the usage and the file access patterns that appeared in the real world. And then based on that, they understood what are the actual design requirements for a distributed filesystem? So they were able to really justify the decisions that they made in Sprite.



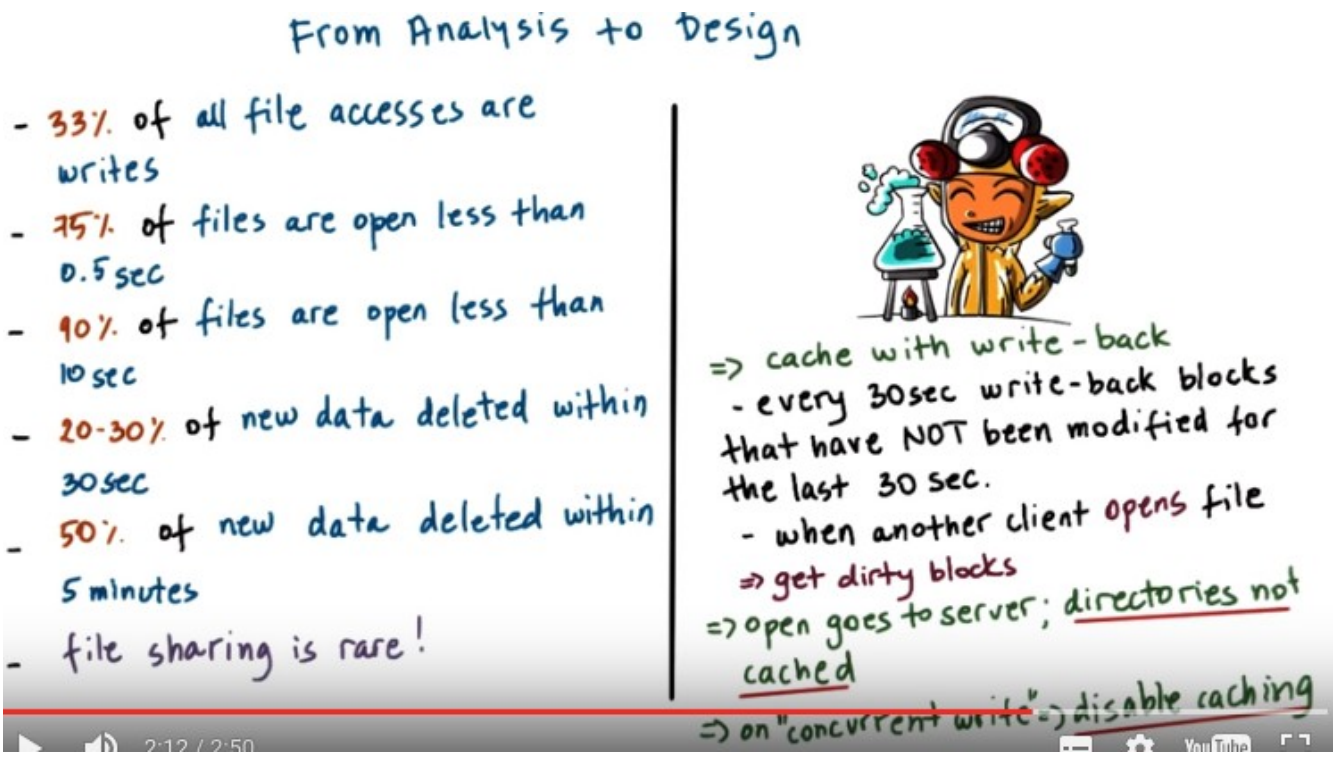
上圖左邊第二三行的數字是 75%, 90%

上圖左邊的內容, 總結起來就是: 1. 大多數文件都只是被讀了, 沒有被寫. 2. 大多數文件都在短時間內被 open 了. 3. 很多 new data 都在短時間被刪了. 4. 很少有 File sharing.

本段的內容, 簡單地說, 就是 caching + session sematics (no need to write-back on close) + support concurrent access (but no need to optimize). 注意這只是本段的意思, 下段講的實際的 decision 跟本段相比, 有小不同.

25. In the paper on caching in the sprite system, the authors performed a study of how are files accessed in the file system used at their department. That was a production system used for all types of tasks. This is what they found. 33% of all of the file accesses are writes. What this means is that caching can be an important mechanism to improve performance. Two-thirds of all of the operations can be improved, but what about updating this remaining one third of the accesses? If we choose a policy that's write-through where every single write goes back to the server, that means that these one-third of the file accesses will not be able to use the fact that there is a local cache on the client side. (以下這句話的意思是指, 正確的做法是 catching + write through + something more than write thourgh): What this means is that caching is okay, it's a useful policy to use in sprite. However, using write-through is not going to be sufficient. They need to figure out what else they can do. So one thing they considered was how about session semantics. We don't have to write-through whenever the file is updated, but when it's closed. However, then they looked at these two numbers and it turns out that 75% of the files are opened for only half a second, just very briefly. And that if you look at what is the number of files that's opened less than ten seconds, we go up to 90% of all of the files. This means that with session semantics, they will need to perform updates to the remote server within less than half a second for many of the files and then within less than ten seconds for really most of the files. So for that reason, session semantics is really not going to solve their problems. They're still going to have

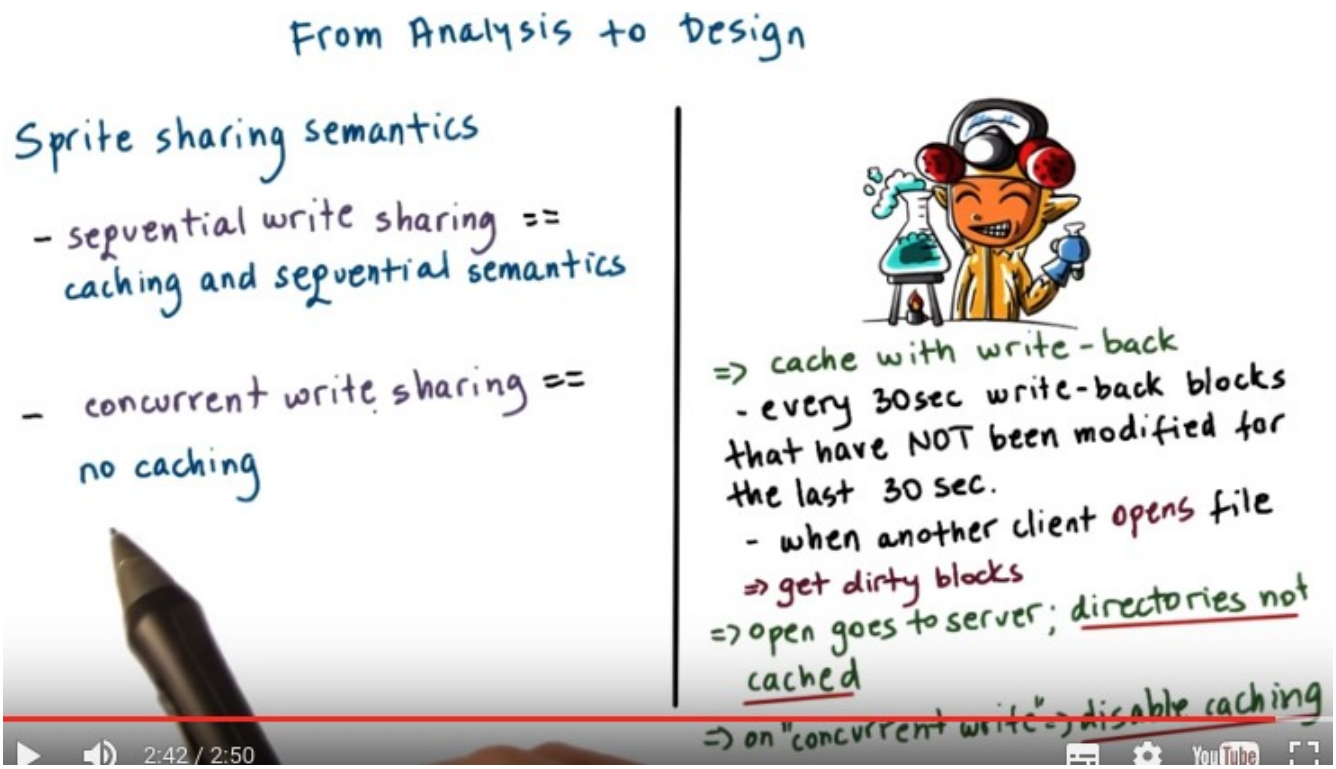
too many interactions with the remote file server and this is what they're trying to avoid. Now next observe something interesting. They realized that a lot of the data is deleted within a relatively short time after it is created. 20 to 30% of the new data is deleted within 30 seconds in their trace. And then for 50% of the new data, they observed that it was deleted within five minutes of being created. And they also observed that file sharing in their system was rare. That the situations in which multiple clients are at the same time working on a file, that really doesn't occur very, very often. So because of these observations, they made first the following decision. A write back on close, which is what appears in session semantics. Well, that's really not necessary. We don't really have two sharing situations and most of the data will get deleted anyways. So forcing the data to be written back to the server when the file is closed, doesn't seem like it will be useful. If the file is deleted, who cares. Now, all of these things are not very friendly to situations where a file needs to be accessed concurrently by multiple clients. However, the fact that they observe that file sharing is very rare, that meant that, that's okay. There's no need to optimize for this kind of situation of concurrent access. However, they did observe some file sharing. It's not like their statement is that there is no file sharing in the system. So because of that, they have to make sure that this distributed file system is useful for the situation when the files are truly shared and it somehow must be supported.



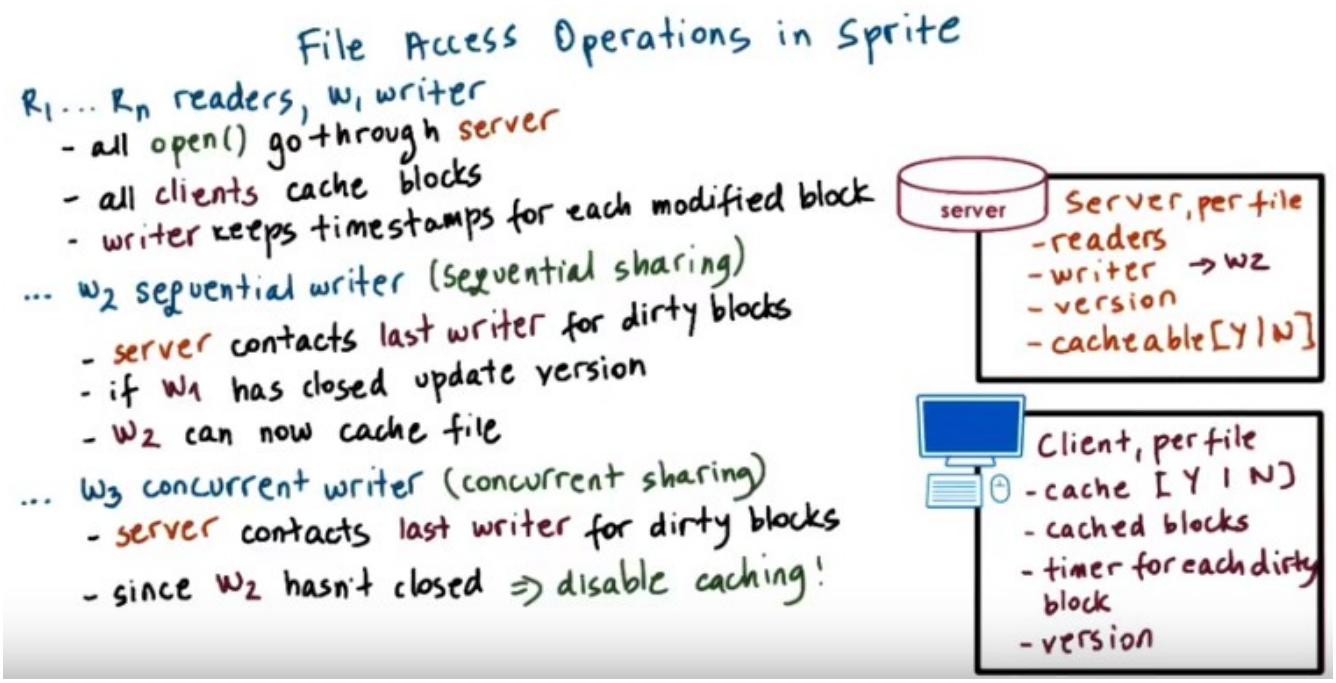
本段的內容, 簡單地說, 就是實際的 decision 為: catching + session sematics (no write-back on close, but write-back every 30 sec the blocks that not modified the last 30 sec, and open goes to server) + support concurrent access (but no need to optimize, and disable caching when concurrent write) + directories not cahced + get dirty blocks.
注意 writer 也是 client.

26. Based on this workload analysis, the author's made the following design decisions in Sprite. First, the Sprite will support caching and it will use a write back policy. First, every 30 seconds the client will write back all the blocks that have not been modified in the past 30 seconds. The intuition behind this is that the blocks that are more recently modified will continue being modified, that the client is still working on that part of the data. So it does not make sense to force to write-back those particular

blocks. Instead, wait a little bit until the client is done working on that piece of the file. And then right them back to the server. This will avoid repeatedly sending the same blocks over and over back to the server. Note that this 30 second threshold is directly related to this value 30 seconds which is their observation that a lot of the data will get deleted within 30 seconds. When a client comes along and wants to open a file that's **currently** being written by another client, the server will contact this **writer** client and will collect all of the outstanding dirty blocks (由於 open 都是 go to the server, 故那個 reader client 就知道這些 dirty blocks 了). In the system, there is **no** explicitly write-back on close, so it's possible that a client has finished working on a file completely, closed the file, and there's still modified blocks in that particular client cache. Note that with this policy, Sprite allows for a file to be open, modified, closed, open, modified, closed multiple times, before any data actually gets written back to the server, and this is one way in which Sprite is able to optimize this 33% of write accesses. Now, note for this to work, every open operation has to go to the server. And what that really means is that the directories cannot be cached on the client (因為 directory 的操作都是 open, 由上句知這些操作都要 go to the server, 所以沒必要在 client 上 cache 這些 directory). So the client can not perform a directories related operation, that looks up a file, and opens a file, and creates a file directly, using its cache only. It has to go to the server. And finally, in the cases where these rare concurrent writes do occur, then Sprite will completely disable the caching for that file. And all of the writes will be serialized at the server side.



In summary, Sprite distinguishes between two situations. When the files are accessed in a way where the writes don't really happen concurrently, instead over time the clients take turns who's writing to the file, Sprite allows caching and provides sequential semantics for the file sharing. In contrast when Sprite determines that a file is shared concurrently for write when multiple clients want to write to that file, then Sprite completely forbids caching. It will disable caching. Because this situation doesn't happen frequently the penalty on performance will not be significant.

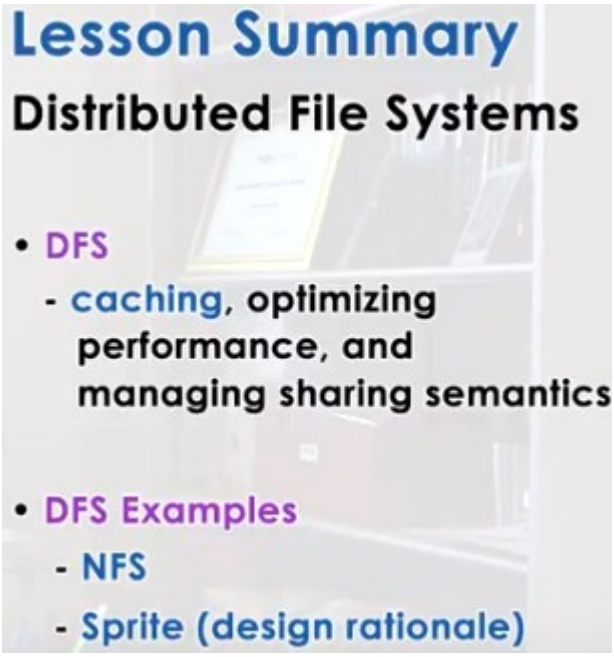


上圖中是 Y | N, 不是 Y I N

本段即前面講的那些原則的一個實例(或應用), 故大多數都是重復前面的. 注意 writer 也是 client.

27. Let's now illustrate the policies that are used in Sprite by walking through what happens in different types of file accesses. In the process, we will also look at what are the different pieces of state that the Sprite distributed file system would have to maintain and the server, and at the client's side in order to be able to maintain the required file sharing semantics. Let's say, initially we have n clients that are accessing the files for reading and one writer client. All open operations will go through the server and the server will allow all accesses. All of the clients, readers and writers will be allowed to cache blocks of this file and the writer client will have to keep track of when was each block of the file modified in order to be able to enforce the write back policy in Sprite every 30 seconds, the blocks that have not been modified in the past 30 seconds. So we'll have to keep track of some timestamps. The Sprite writer can keep closing the file and then deciding to reopen it to modify it some more. When it decides to do this, the contents of the file are cached locally in the writer's cache, but the open still has to go to the server. At this point, the writer will need to check quickly with the server whether its cached value is the same as what the server sees. And because of that, they'll need to keep some sort of version number. To support these operations, the client would need to keep track of some information for each file. This includes some status whether or not the file is in the cache or not, then what are all the cached blocks from that file. For all of the dirty blocks, when were they modified last. So that we can run the write back mechanism and then also version number. The server will also keep track of some information for each file, like what are the current readers, what is the current writer of the file, and

also what is the current version of this file. Now let's say, at some point after the writer W1 has closed the file, another writer W2 shows up. And this is what you refer to as a sequential writer. They're not trying to write the file at the same time. When a situation like this happens, this is what we refer to as sequential sharing. What needs to happen in that case is the server needs to contact the last writer that it is aware of. In this case, that's going to be W1 and to gather all of the dirty blocks and it's very convenient that W1 keeps track of the dirty blocks. If W1 has closed the file, the server will update. The new version will update the new writer. The W2 is the current writer of the file and at that point, W2 can proceed and it can actually cache the file. And now while W2 is still modifying the file, it still has the file open and is accessing it and it's writing to it. We have that unfortunate situation that rare situation where W3 appears and it wants to concurrently at the same time write to the file. So this is what Sprite refers to as concurrent sharing. There are two writers, W3 and W2 trying to write to the file at the same time. When the write request comes, the server will just like before, contact the last writer to gather the dirty blocks and that's going to W2 in this case. Now, once the server contacts W2, it will also realize that W2 hasn't actually closed the file. What will happen in that case is that W2 will write back all of the dirty blocks and then the server will mark this file as it is not cachable, it will disable the caching of this file for everybody. Both W3 and W2 will continue to have access to the file, except they will not be able to use their client's side caches and all of the file accesses will have to go to the server. For this reason, it makes sense on the server side to maintain some information for each file whether or not the file is cachable. When one of the two writers, W2 or W3, closed the file, the server will now see that close operation, because every single operation in this case will go to the server. The file is not cached, otherwise. When the server sees that one of the clients closes the file, at that point, it will change this cacheable flag to point to yes, the file is cacheable. And so the remaining clients (包括 writer) can start caching the file and can enjoy the performance optimization of caching. So, one unique feature of Sprite is that it dynamically enables and disables caching depending on whether or not there is a sequential write sharing versus concurrent write sharing among the clients in the system.



28. In this lesson we looked at distributed file systems and described the use of caching and accompanying mechanisms to provide optimized file access performance on one side. And also to maintain meaningful file-sharing semantics on the other. We looked at the specific decisions made in NFS and the sprite distributed file system. And for sprite we looked how its design was motivated by

certain observations regarding file usage and file access patterns.

29. As the final quiz, please tell us what you learned in this lesson. Also, we would love to hear your feedback on how we might improve this lesson in the future.