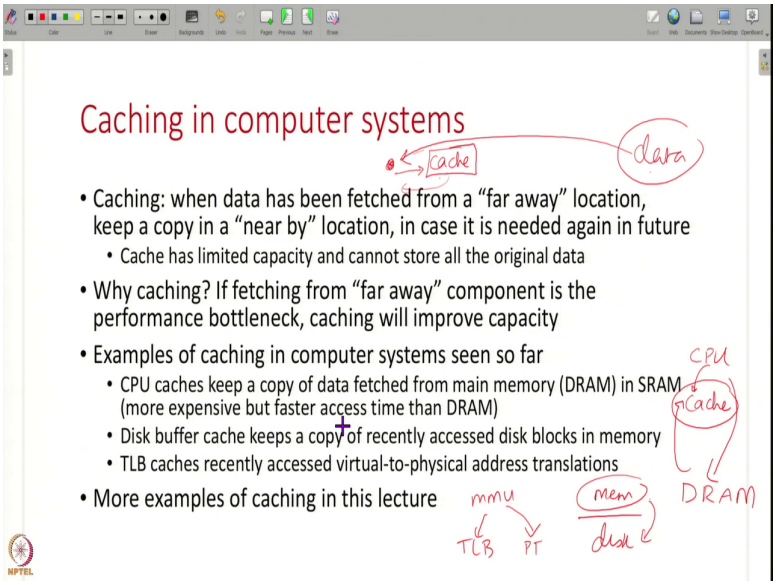
Design and Engineering of Computer Systems
Professor Mythili Vutukuru
Computer Science and Engineering
Indian Institute of Technology, Bombay
Lecture - 48
Caching

Hello everyone, welcome to the 34th lecture in the course design and engineering of computer systems. In this lecture, we are going to study more about the concept of caching. In the previous lecture, we had said that if IO is your bottleneck of fetching data from some devices your bottleneck then caching is a common technique that is used to improve performance. In this lecture, we are going to study more details about what is caching and how do caches work.

(Refer Slide Time: 0:47)



So, the basic principle of caching is you have some data that is located at some, so called far away location and you need it here, then if you have to fetch that data from that far away location all the time, then that is going to impact your performance. So, for example, whether it is disk or some other node or something. So, then what you do is you store the recently fetch data in what is called a cache in a nearby location. So, that the next time you need the same data again, you do not have to go all the way here, you can simply get it from the cache.

So, a cache, note that it has limited capacity. If the cache could accommodate all this data, then why not keep everything here itself. Now, that will solve your problem. But typically, caches are only used to store the recently used data because they can only store very limited amounts of data, not all the original data. And the fetching from this far away component if

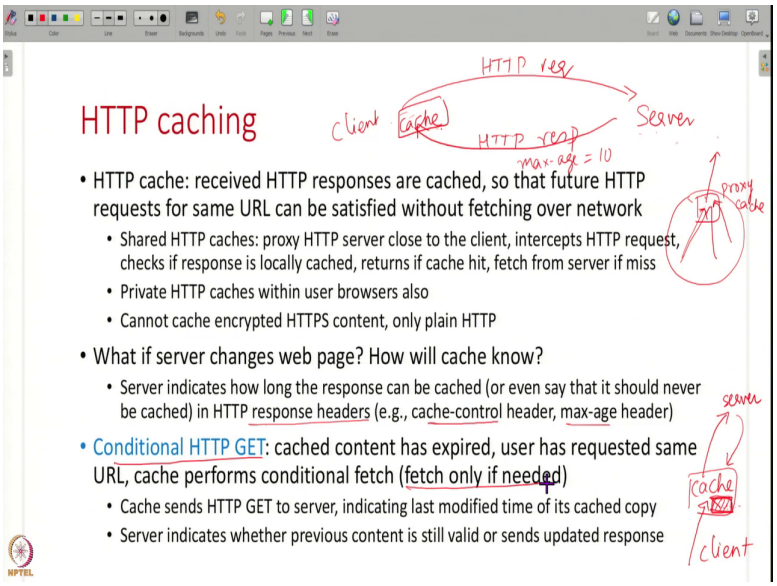
this is your performance bottleneck, only then will a cache improve your performance. If something else is a performance bottleneck, adding a cache may not be useful.

But if this IO is your performance bottleneck, then instead of most of the time, if you are just fetching from the cache and not going all the way here, it will improve your systems performance and capacity. So, we have seen many examples of caching multiple times in the course so far. And across all of them, the principle is the same, you have CPU caches, the CPU needs some data from DRAM, whether it is instructions or program data, the CPU needs it from DRAM, and instead of going to the DRAM all the time, the CPU has many levels of caches, and it will store the data from DRAM into caches, so that in the future, it can directly access data from the cache.

Similarly, you have the disk buffer cache whatever you read from disk, you will store it in memory in the disk buffer cache, so that you can avoid going to the disk most of the times. Similarly, TLB caches, they store the recently accessed virtual to physical address, so the mmu will check the TLB cache first. And if it cannot find the address, translation, the TLB cache, then it will go to DRAM and walk the page table. So, these are all examples of caching we have seen so far.

So, in this lecture, we are going to see more examples of caching and more importantly, we are going to try and come up with some general principles to design caches so that you can design them in any system.

(Refer Slide Time: 3:14)



So, one popular cache is what is called HTTP cache. So, whenever you make an HTTP request whenever the client makes an HTTP request to some server, and gets some HTTP response back whether it is a web page or something like that, you get an HTTP response back, what you can do is you can store this response in a cache, so that in the future, if this client or somebody else wants this same URL, again, the same web page again, you do not have to go to the server, you can simply get it from your HTTP cache.

So, HTTP responses are frequently cache. So, these can be shared caches for an organization, you can have something what is called a proxy server, so that all clients in the organization will go through this proxy server to access the internet. So, that if some web page is already cached at this proxy server, then if suppose one client access the webpage, and it is got and stored in this proxy cache, then other client wants to access the same web page, it can simply get it from this cache, you do not have to go all the way to the server.

So, you can have such shared caches for multiple users in an organization. and this cache, this proxy server will, if it is not there, this proxy server will get it from the remote server for you and serve it to the clients. Or you can also have private caches within browsers also just for one user, so that if the user wants to access the same page, again, immediately, it is available, these are not shared with anybody else. So, all of these types of caches are frequently used in computer systems today.

Note that one thing to remember is you can only cache plain HTTP, you cannot cache encrypted HTTPS content that easily because you do not know what is inside that HTTPS content. So, that has to be remembered. So, now then the question comes up I have stored some webpage in a cache, what of the server changes the web page. It has some news website, the news has been updated. I do not want to be looking at this old stale news all the time when the news gets updated, I want to access that fresh news, not this cached old news, then how does caching work in such cases, so normally, what happens is the servers will indicate how long this caching can happen.

So, when the server sends a response back, it can say the maximum age of this responses, say 10 seconds or something. So, the server will say, for the next 10 seconds, you can use this response. But after that, I may update this response. So, therefore, do not use it after some maximum duration. So, there is an expiration date that the server puts on HTTP responses, or it can also say there is something called cache control header in the HTTP response, where

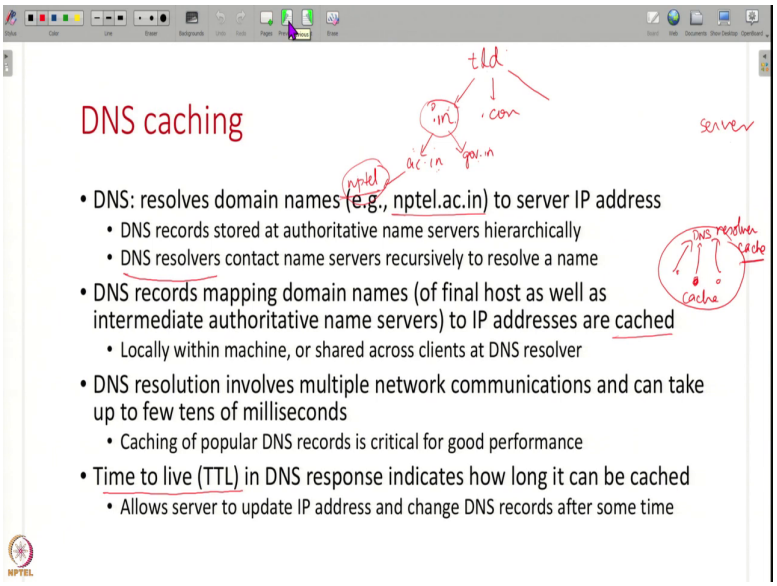
the server can say, do not cache this at all, I am constantly updating this page, please do not cache it.

So, there are various HTTP response headers, like cache control and max-age that the server will set. The HTTP response consists of all these headers and the actual content. So, these headers will be set to control the behavior of the cache. And what these caches will do is if the cache has some expired content, the user, the client has requested some web page and the cache sees that within its storage, it has some expired web page, then maybe what the server may not have updated this, this might still be the valid webpage.

So, how will the cache know what it will do is it will talk to the server and do what is called a conditional get. That is, it will fetch the page from the server only if required. So, it will tell the server, hey look, I have a copy of the page that was last modified at so and so time, it will tell the server some information about its cached copy and ask the server is this the most recent one or do you have an updated version?

If the server has an updated version, it will send the updated version otherwise it will send the response indicating what you have is okay. So, there are ways in HTTP protocol to do this conditional HTTP gets where you will fetch HTTP response only if your cached copy has expired. So, HTTP caches frequently do this, it all of this is done in order to avoid going to the server multiple times because the server might be far away, and it might be slow and hurt your application performance.

(Refer Slide Time: 7:47)



So, the other type of caches are DNS caches, we have seen that DNS is a way in which a domain name for example, like nptel.ac.in is resolved to the IP address, so that your client can actually send a data to the particular server IP address. And these DNS records are stored at authoritative name servers that is you have a top level domain. And then under this, you have name servers for dot in, then you have dot com various top level domains, and under this dot in, you have ac dot in, which is academic websites, and under this you have nptel.

So, you will first talk to this authoritative name server, the top level domain, get the IP address of this server, then you will talk to the server get the IP address of this server, then you will talk to this server and get this IP address. In this way, DNS resolvers do this hierarchical name resolution to find the final IP address. Now that you have found out this IP address after this long process, you do not want to do this all the time. Therefore, these DNS records that map from a name to an IP address, these will be cached.

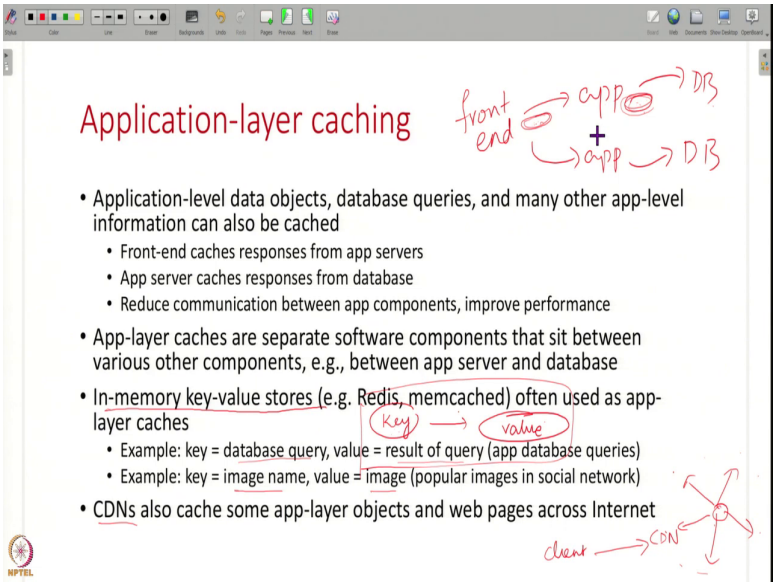
They can either be cached locally in a machine or once again shared at the DNS resolver. If there are multiple clients in an organization, all of which are using the same DNS resolver, then this DNS resolver can itself cache these DNS records or each machine can also cache the DNS records. So, the DNS records not just this final resolution, but you learned of the IP addresses of many intermediate nodes in between also, so all of these can also be cached.

Because tomorrow if you want some other dot in, government.in, then you have cached the IP address of this name server, you can directly go there instead of starting your resolution from the top. In this way various intermediate IP addresses as well as the final IP address. All of these DNS records are cached. And caching is very important because as you can see, DNS resolution involves multiple network communications and it can take a few tens of milliseconds. So, therefore, caching is very critical.

Now again, the question comes up what if this IP address is updated, and I have some value in my cache, but the actual servers IP address is updated, then will I suffer by using an old version of the cached DNS record. That is why DNS responds once again, for any cache, you will have this time to live there is a field in a DNS response called time to live, that the server will set saying, this IP address is valid for so long.

After some time, you have to once again do this resolution and fetch the latest DNS record. So, this time to live field allows us to update the DNS record after some point of time.

(Refer Slide Time: 10:44)



So, apart from HTTP and DNS caching, in general, any application can do its own caching. For example, we have seen this web application architecture where there is a front end, there are various application servers, there are various databases. This is how a typical application looks like, a real life application. So, at every stage, you can have caches, for example, the application server if it is sending the same queries to the database all the time, it can just have a small cache of recently received database queries, the front end can keep a cache of the responses received from application server.

So, if some user has just searched for a certain product, and again, another user searches for the same product, then you can simply send the cached response back. So, in this way, between any two components, you can have a cache to reduce the communication between these components and improve performance and anything application level data objects, database queries, anything can be cached. And all of these caches are, again, separate software components that sit between various other components.

So, you are adding extra complexity into the system with caches because you are bringing in new components into the system, but they improve performance. And frequently, for these application layer caches, you use what are called in memory key value stores that is simple databases that keep a mapping from some key to some blob of data. For this key, this is the data, this key, this is the data. So, such stores of key value stores, and they store them in memory.

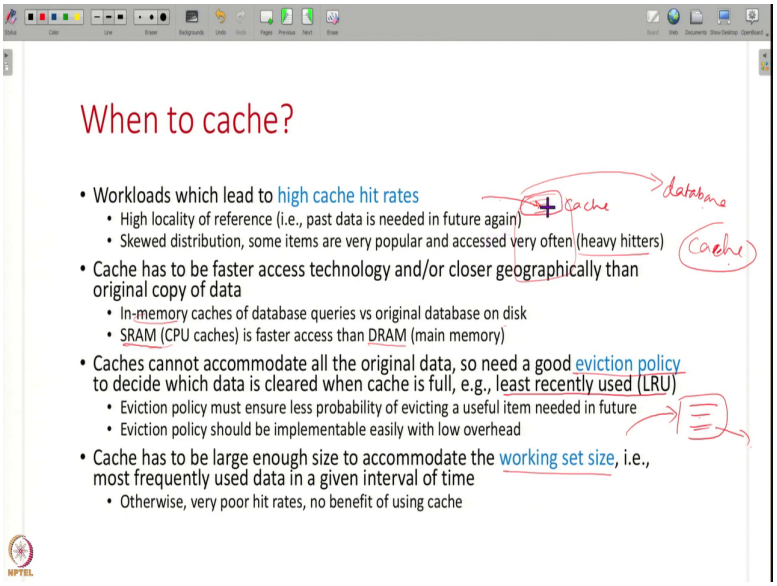
So, there are popular software like Redis, memcached that serve as in memory key value stores, and these are often used as caches. Note that it makes sense to keep cash in memory. I mean, if you are going to disk and you are bringing something again, storing it in cache on disk is again, slow. So, the whole point of cache was to quickly access it. Therefore, usually, these caches store all of these recent application data in memory. And examples are, for example, if you want to cache the recently obtained images, there is an image database, you are getting some images.

So, you use the key as the image name, the value is the image contents of the key is your database query. And the value is the result of the query, all the rows of the database table that were returned. So, in this way, use some key to identify the value. The next time a query comes, a request comes for the same key instead of going all the way to the next component, you can simply look it up in the cache.

And we have also seen other examples like CDNs whenever there is a web server has some web pages that will push this content out to CDN, content distribution networks, which have replicas throughout the globe they are geographically spread, so that any client can instead of going to the server, it can directly fetch a web page from the CDN itself. So, CDNs also cache various application layer objects, web pages, images, so that clients do not have to come all the way to the system, they can directly get the data from the CDN.

In this way, at every level between application layer components between clients and the system in the form of CDNs in the form of HTTP, proxy servers, DNS caches all through, you have various levels of caching happening in computer systems so that you can avoid this extra expensive communication. Now, across all of these caches, there are certain common design principles and common things to keep in mind that I would like to point out in the next few slides.

(Refer Slide Time: 14:27)



So, the first question is, when do you cache just because you are fetching something from another component, you do not just put it in a cache? There are certain guidelines to help you decide, is this worth caching? Is it worth putting an extra cash component between these 2 components or not? And what are these guidelines? So you will only cache when the workload will lead to high cache hit rates. For example, you have access to some piece of data from some remote database.

And then this data will be needed again immediately in the future. Only then it is worth putting it in a cache when there is high locality of reference, suppose you have access some data, you will never again use it in the future or very less likelihood that you will use it in the future. There is very low locality of reference, then there is no point caching it. Similarly, it is worthwhile caching when there is a skewed distribution that some items are very, very popular, some images are being repeatedly fetched from the database, then it is worth putting them in the cache.

But if once you access an image, you will never use it again, then what is the point? such popular items. If there are heavy hitters, then you can cache it. So, you have to think through does your workload lead to high cache hit rates or not. If I want to you take the extra effort and put a cache will I get good performance gains or not only then you will cache. The other thing is the cache has to be faster access then, or somehow closer than the original copy of the data.

If your cache is also here as far away as your original data or it is slower than accessing the original data, then what is the point, there is no point in caching. Therefore, usually caches are in memory caches, so, that instead of going to disk, you can get it from memory. CPU caches are a different kind of technology called SRAM that is much faster than the DRAM that is used for main memory. So, in this way, caches are useful only if they are faster and or closer. Then the other thing caches need is you need a good eviction policy.

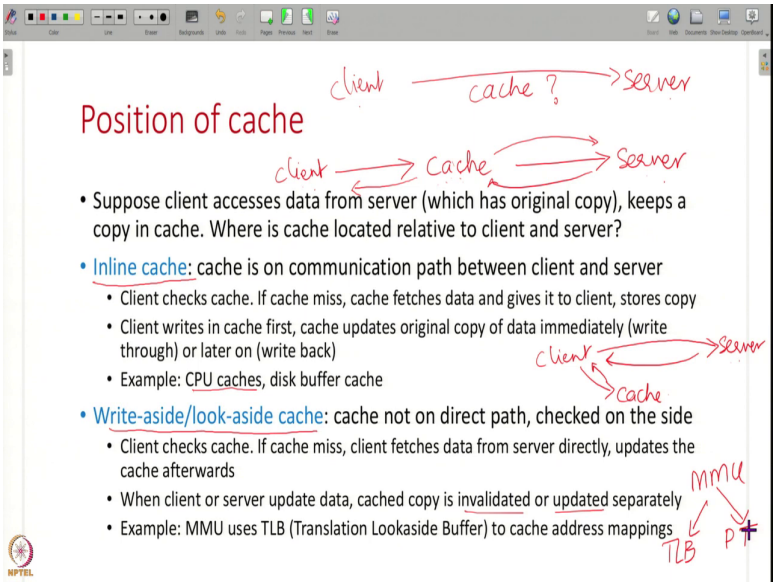
Of course, if your caches big enough to store all the data, then why not just store all the data here. That will never happen. Because your caches are somehow different technology, they are more expensive, you can never store all the original data, you can only store a subset of it. Therefore, which subset of the data will you store? What if the cache is full, I have to throw away some older item. Therefore, you need a good eviction policy, which will help you decide. Once my cache is full, then I want to store another item, I have to throw away an old item.

So, your cache should be in such a way that you can implement a good eviction policy, which will ensure that you do not want to evict a very useful item. As soon as I throw this away immediately. If somebody asked for it, what is the point, it is a bad eviction policy. So, my eviction policy should be good, like LRU, Least Recently Used as a common eviction policy used in all of these caches something has not been used for a long time, most likely nobody needed in the future, again, let is throw it out.

And this eviction policy, you should be able to implement it easily without too much overhead, for example, finding the least recently used item should not take like10 milliseconds, then it is very inefficient. So, things like that your eviction policy should be good. And the other thing is your cache has to be large enough to accommodate the working set size. So, suppose the application is using some subset of this data frequently, that data should be accommodated. If your cache is very small, whenever you want anything in it, it is not there, it is not very useful.

So, there are all of these guidelines that tells you for a given application for a given workload for a given system, is it worth using a cache or not? Now the other thing is, where is this cache located?

(Refer Slide Time: 18:17)



Suppose you have a client, and you have some server from which you are fetching some data, which has the original copy of the data that these are general terms. The client can be the CPU server can be DRAM client is basically something that is requesting data from a server. And now you have a cache in between. So, the question is, where is this cache located relative to the client and the server. So, there are two types of caches. One is what is called an inline cache. What is an inline cache?

An inline cache is a cache that sits directly on the path between a client and a server that is between the source of the data and whoever wants the data that is an inline cache. What does this mean, the client will never directly talk to the server, the client will always check the cache. If the item is there in the cache, the cache will return it. If the item is not there, if it is a cache miss, then the cache will talk to the server, fetch the item and then give it back to the client.

So, the cache is always in the middle that is an inline cache, the client will never directly talk to the server. Examples are most of the caches we have seen so far, like the CPU cache is an inline cache, the CPU will always check the cache first and then whatever has got from memory is put into the cache and then returned back to the CPU. Then the disk buffer cache all of these are inline caches.

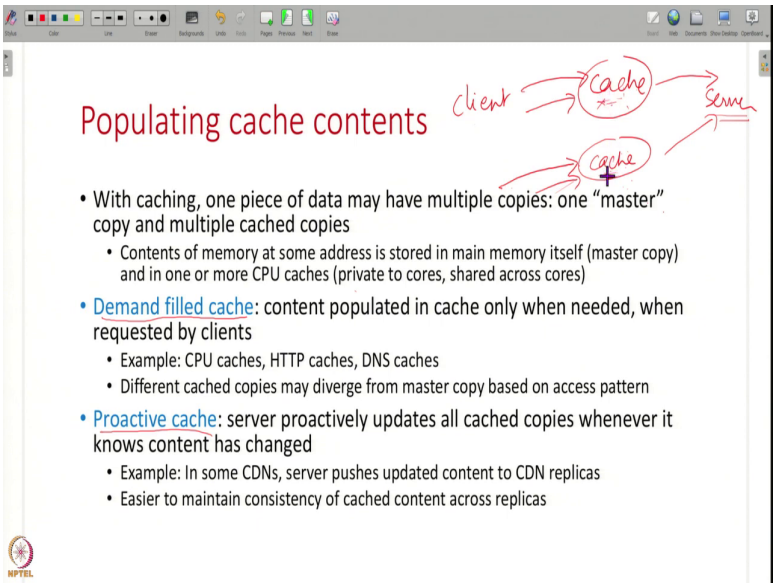
The other kind of cache is what is called a write-aside or a look-aside cache. That is the client will first check the cache and if the item is there in the cache well and good, if the item is not there in the cache, the client will directly talk to the server and get the data item. And later on

update the cache. So, the cache is not in the middle, it is on the side. It is not on the direct path, the client and server can directly talk to each other, the cache is only on the side, and the client or the server, they will update the cache whenever some data changes.

So, for example, if the server changes a data, it will tell the cache, hey you update your copy or invalidate this cached copy has to be either marked as invalid or it has to be updated. Similarly, the client can update the data at the server and then the client will tell the cache, hey, update your copy. But the cache is not involved in this communication between the client and the server. So, what is an example of look-aside cache? We have seen one before the TLB, if you remember, so the TLB is sort of on the side, the mmu will check the TLB.

And if it is a TLB, means the mmu will directly access the page table, get the address translation and then update the TLB. So, the TLB is a look-aside cache. That is why it is called a translation look-aside buffer. It is not on the path between the mmu and the page table. So, these are just two types of caches. When you look at a cache, you should be able to understand what type of a cache it is.

(Refer Slide Time: 21:17)



Then the next question comes up, how do you populate the contents of the cache? You have a master copy of the data somewhere at the server and some clients need this cached copy, how do you populate this cache? So, again, there are two ways. So, there is what is called a demand filled cache. That is the content is populated only when needed the client is there, the client will request some data, only then the data will be put into the cache only then the data will be fetched from the server and put into the cache that is a demand filled cache.

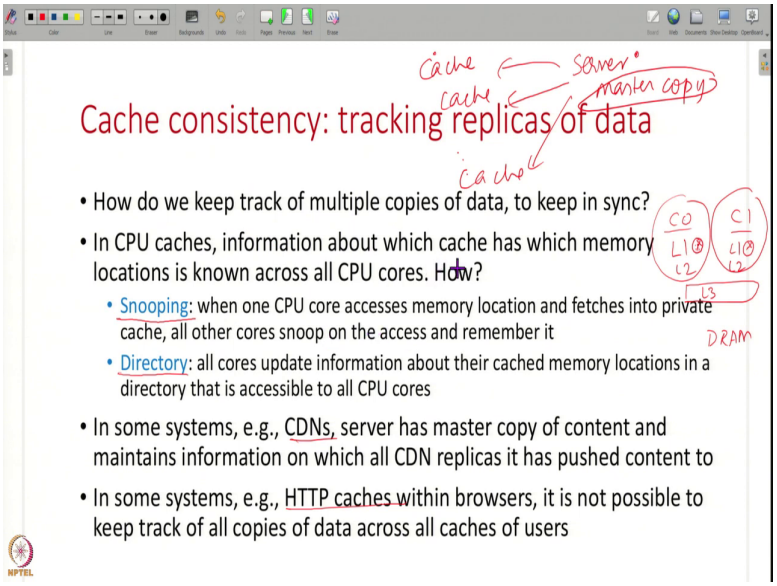
That is the client will pull the data when needed into the cache. Things like CPU caches HTTP caches are all demand filled caches. And when, with demand filled caches, it can so happen that if you have multiple caches, this client is requesting some data and this client is requesting some other data, then these copies of these different caches might diverge. For example, the CPU core has requested some memory locations, they are there in this cache, the CPU core has requested some of the memory locations, they are there in this cache.

Now the CPU core has updated some memory location, but that will not automatically come here. Because why it is a demand filled cache only if the CPU core request it will come into this cache. So, if you have multiple caches, the contents of these caches might diverge in demand filled caches, because based on demand, only, you are updating it, you are not generally all the time updating the cache contents. And then you have proactive caches, which is the server will actively push, whenever any data changes, the server will actively take responsibility to populate all of these caches and keep them consistent.

For example, in some CDNs the server has distributed its files to CDN the server will update the CDN replicas whenever a web page changes, it will push it to the CDN replicas. So, in such proactive caches, it is easy to maintain consistency across these multiple caches. But otherwise, if it is a demand filled cache some CPU core has updated some item here. So, then the CPU core may not have the updated copy of that item, because it may not even have the data in its cache line.

In the case of CPU caches, when you have private caches to every core. In such cases, if its demand filled, it is harder to maintain this consistency.

(Refer Slide Time: 23:43)



So, now the important question with any cache, once you have multiple copies of data you have your server that has the master copy of the data. And then you have multiple caches, throughout and each of this is caching some subset of this data, then the question comes up, how do you keep all of these cached copies of the data in sync? If some data changes here, it has to be updated in all the caches. If the server changes some data, it has to be updated in all the caches.

Otherwise, if you have old values of data in these caches, then it will be incorrect for the application to use an old value of data. So, with caches, this challenge comes up that now instead of one copy of data, you have created multiple copies of data all through the system. And somehow you have to take the responsibility to keep all of these multiple copies of the data in sync. And that problem is what is called cache consistency.

Anytime you have multiple caches and clients are accessing data from multiple caches, you have to ensure that everybody accesses the same consistent version of the data. So, the question comes up, how do we guarantee this cache consistency? So, we will see some of the ideas to do that. So, let us take the example of CPU caches. Now this is something we have seen a lot.

So, in the CPU, if you have two different CPU cores, each CPU core has its own some private caches, like the L1 cache, L2 cache, they are private to this core. And then you have a common, this L3 caches, a shared cache, and then you have DRAM, you can have some

structure like the some caches only this core has, some caches are shared across cores, and so on.

So, now, across all of these caches, you have to maintain information. For example, if this cache has certain memory location X stored here, and the CPU has updated this memory location X, then when the CPU also wants to access X, it needs to know the information that oh, no, the updated value is here. So, this information of which cache has which item, this needs to be tracked. And how was this tract there are two ways.

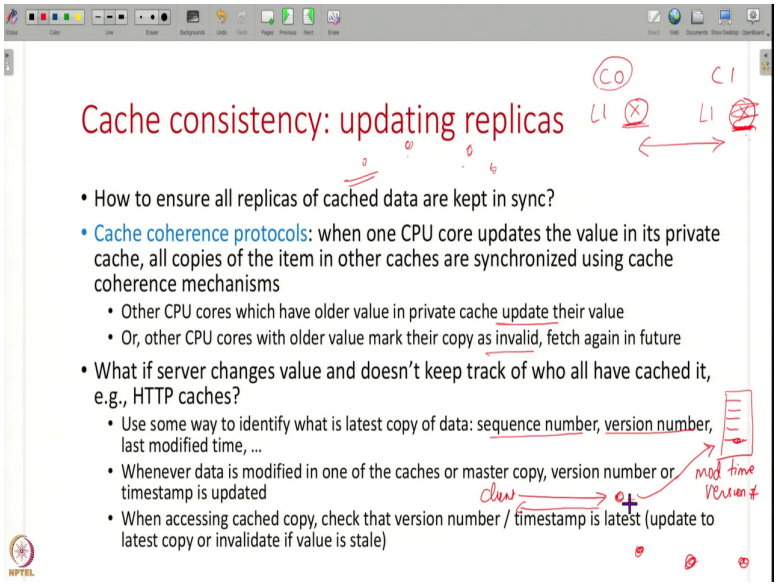
One way is what is called snooping. That is, whenever a cache gets some data item, all the other caches also snoop Oh this guy has obtained this memory location from RAM, this core has obtained this memory location from RAM, so everybody is snooping around to see what the others are doing. And based on this, everybody knows, now this core knows Oh, in the past, I have swooped, I found out that C0 has this memory address X. So, the next time I wanted, I should check with C0 for its latest copy.

In this way, you can snoop or you can maintain a directory that is all CPU cores can keep a directory which has information about which CPU core has cashed which memory locations. So, these are two common techniques used at the level of CPU caches to keep track of which cache has which memory location. And in some systems, of course, it is easy to keep track. For example, in CDNs, the server has all the master copy of the content and the server is distributing you are entering into a contract with a CDN provider and saying, Oh, please cache this content in all of these different places.

So, it is easy to know, because you are the one who is actively pushing content out to CDN. So, it is easy to keep track of which CDN replica has which webpage. And in some systems like HTTP caches, it is very impossible to know if people are caching web pages in their browsers, it is very impossible for the web server to look at the caches of all the billions of internet users and see who has this webpage and whose cache.

So, therefore, in some systems, like HTTP caches, it is impossible to keep track of all the copies of the cache data. But in some systems like CPU caches, you have to keep track of which cache has a which memory location so that you can consistently access the latest copy. Now the first problem in cache consistency is keeping track of the replicas.

(Refer Slide Time: 27:50)



The next problem is updating all the replicas, how do you keep these replicas in sync with each other? So, that is what cache coherence protocols, so now when one CPU core, the CPU core in its L1 cache has cached a memory location X, a cache line X, and this other CPU core also in its L1 cache, also it wants to access the same memory location X, then what will you do? Then there are some cache coherency traffic that runs between these two CPU cores in order to synchronize the value of this memory location.

So, whenever core C0 updates this memory location, then C1 will either update its value also, or it will invalidate X value, it will say, fine, let me delete this thing from my cache. So, when one CPU core updates a value in its private cache, all other copies of that data and all other CPU cores have to be synchronized in two ways. Either all the other cores also update the value they will also write the value and updated to the latest value pushed by C0, or they will invalidate that value, they will mark it as invalid.

And say later on, if the CPU core request the same memory location, then I will fetch it, I will not worry about it for now. So, you have to keep these multiple replicas or for data if a data item is cached at multiple locations, if one copy is updated, all the other copies also have to be kept in sync, they all have to update or at least they have to invalidate, they have to remove it from their cache, so, that the next time they need it, they will get the latest value and not use their old stale value.

Now this you can do in CPU caches, because you have kept track using either snooping or directory you know if this memory location is there in all these caches, what about things like

HTTP caches where you do not know where an item is there a web page in which all browser caches it is there all around the world, a web server does not know cases when a web server changes a web page, then how do we ensure that all the replicas are kept up to date, it is very hard. Therefore, in such cases, what we do is, we usually identify the latest copy of the data in some way using some sequence number, version number.

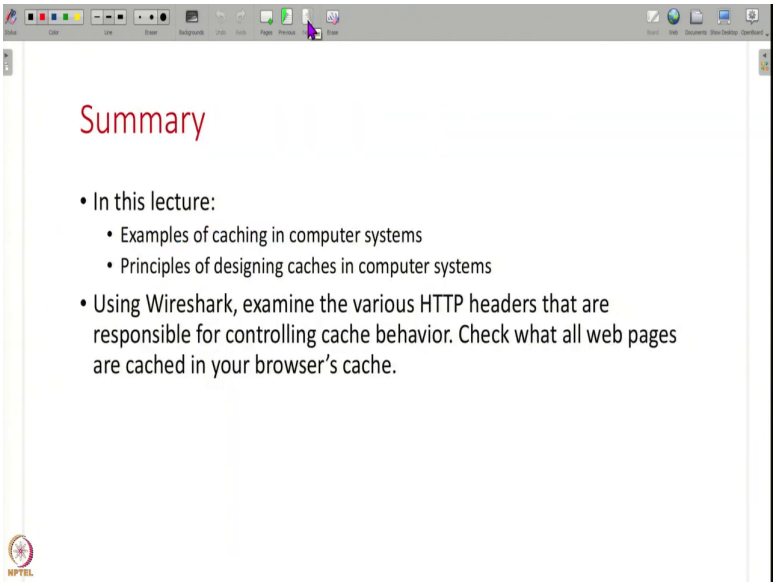
So whenever the web server updates a webpage it has made some changes to a webpage, it keeps some time the modified time the last modified time or some version number, something you will maintain for the data item. So that when there are multiple caches that might have older versions of this data item, when a client requests some HTTP cache, then this cache can go check with the server saying, hey, I have this version of the data or I have a web page with this modified time is that what you have also?

If the server also has the same latest version, then this cache can simply return its value, or if the server has updated, then the cache will get to know that, what I have is the old web page, the newer web page has a higher version number or a later modified time, let me fetch it. In this way, you need some way of identifying this is old data, this is fresh data. So, that when the cache is of course, the server cannot push these updates to all the caches all over the world.

But the caches themselves can check, oh, this is my version number. That is the latest version number, let me update. In this way, you have to put some kind of identification number like a sequence number, version number, timestamp something on these data items, so that whenever caches have to access the same data item, again, they can compare is their version number, is their timestamp the latest or not.

And if it is not the latest, they can either update their copy or invalidate their copy or something like that. In this way, these are some of the techniques available to maintain cache consistency.

(Refer Slide Time: 31:56)



So, that is all I have for this lecture. In this lecture, what I have shown you is many different examples of caching in computer system, from CPU caches to disk buffer cache to TLB, to HTTP caches to DNS caches, application layer caches, we have seen many different examples. And in addition to that, we have also seen some common design principles across all of these caches, like where is the position of the cache it is? Is it inline or is it on the side? How was the cache filled? Is a demand filled or is it proactively filled? How do you track all the replicas of cached content? How do you maintain consistency across cached content?

So, we have seen some common principles that are widely used across all of these caches. So, the next time in a computer system, you have to design a cache, keep all of these in mind and see which of these patterns will fit your requirements when you are designing a cache. So, one exercise for you to do is you know, you can examine HTTP headers in Wireshark and observe these last modified time cache-control, max-age, there are several HTTP headers that are used that decide which HTTP pages which web pages get cached in your system and which do not.

So, observe these things to understand how web servers today control caching in HTTP proxy servers and HTTP caches. So, thank you all that is all I have for this lecture. Let us continue this discussion in the next lecture. Thank you.