Lecture 20: April 11

Lecturer: Prashant Shenoy Scribe: ***Gayatri Belapurkar(Spring 2022), Mikayla Timm(Spring 2019)***

In this lecture, professor starts a new topic "Distributed Web Applications".

20.1 Traditional Web Based Systems

The web is basically a client server based global distributed system. The clients tend to be web browsers or application on the phone or other devices, which access the server component over HTTP. HTTP is a request response protocol, as also seen in Lab 2.

Figure 20.1 shows a basic client-server request response protocol, that exchanges static web pages. But the server side can consist of a complicated server that can process requests. In this image, the browser sends an HTTP request to the web server. The server fetches the document from database and sends back the response to the browser.

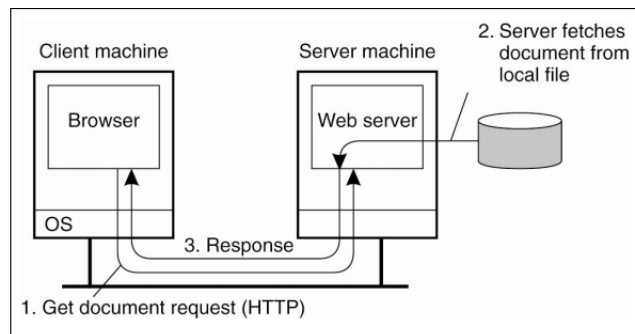

Figure 20.1: Overall Organization of a Traditional Web Site

20.2 Web Browser Client

Browsers are complex with many built-in functionalities. Web browsers have a user interface where the user can submit a request, the browser connects to the server, fetches a web page and renders that web page. As in the figure 20.2, there are multiple components such as user interface, browser engine (part that retrieves the content), rendering engine (take the content, decides the layout, etc.) and other components for network communication, client side scripts for interpretation (example: Javascript interpretation), and a parser that can parse HTML.

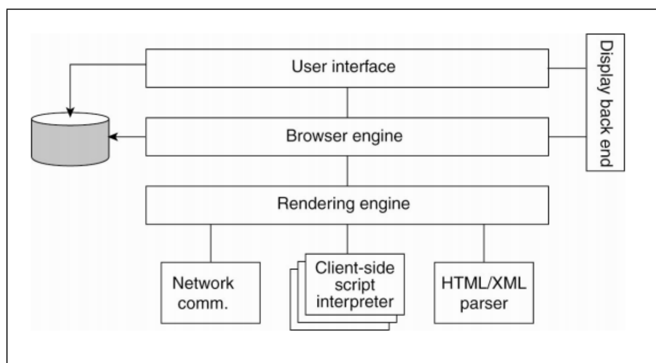

Figure 20.2: Logical Components of a Web browser

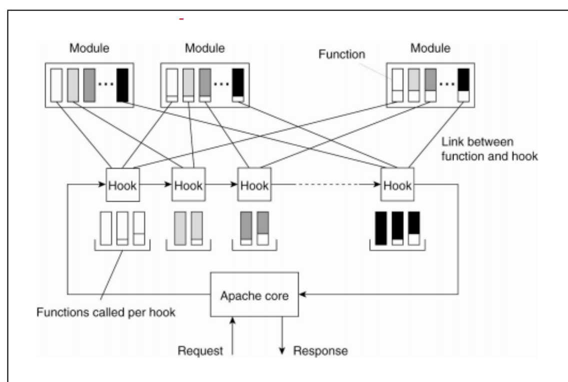

Figure 20.3: Overall Organization of Apache Web Server

20.3 Apache Web Server

The server also has complex components. Figure 20.3 is a Apache Web Server, one of the most popular web servers. The original version is a multiprocess model, other variants also support multithreading. By default, it uses the multiprocess model when starting up. The main process listens on HTTP and assigns incoming requests to child processes. Similarly, the multithreaded model can also be implemented with dynamic thread pools.

The architecture shows the process that takes place, irrespective of whether it is multithreaded or multiprocess. It has a modular architecture that uses pipeline processing. Processing is done by the various modules present (HTTP, SSL, etc.). These can be plugged in when the server is started. We need to configure a set of modules that will perform request processing. When a request comes in, it will go through a series of processing steps. Each module performs partial processing in a pipeline fashion. You can turn each module on or off. The only processing that is supported by the server is if we write the application using PHP. Other languages such as Java, Python will be processed outside the HTTP server.

20.4 Proxy Server

Proxy is an intermediary between the client and the server, making the client-server architecture a client-proxy-server architecture. Proxies can be used for multiple purposes. The example shown in figure 20.4 does protocol translation, converting from HTTP to FTP. Proxies are also used for web caching. A proxy is closer to the client and can process user requests faster than the actual server. Recently used web pages are cached. The browser will send requests to the proxy instead of server. Proxy will process the request. If the requested content is cached on proxy, it is will send back reply to client. Else, the proxy makes another request to the server.

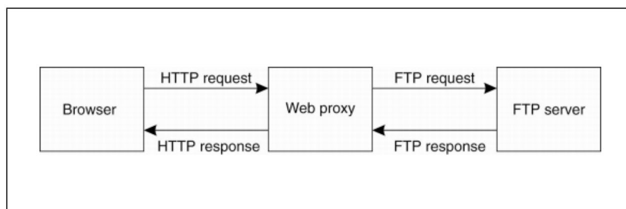

Figure 20.4: Web Proxy

20.5 Mutlitiered Architecture

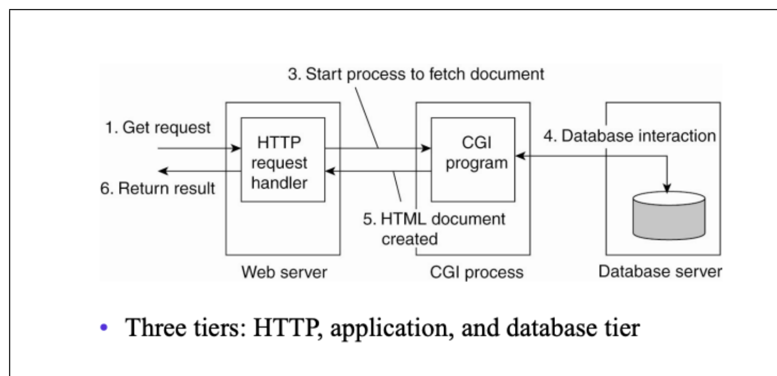

Figure 20.5: Multitiered architecture

The figure 20.5 represents a standard 3-tier web architecture wherein requests are sent from the web server, processed by the CGI program (which is now replaced by Python, Java, etc.) and then to the database. The request is processed and the response is sent back to the client. In this case as well, a proxy can be introduced to avoid repetitive computation.

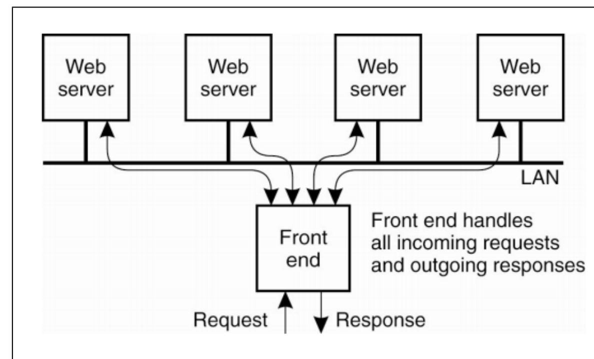

Figure 20.6: Web Server Clusters

20.6 Web Server Clusters

Figure 20.6 shows an example of clustered architecture. Each box in itself is multi-tiered. Each incoming request can be forwarded to one of the four Web Servers. The front end handles all the incoming requests and the outgoing responses. Responses flow back through the load balancing switch. This is a common way to scale applications. If a single machine is not enough, we replicate the web application. The clusters can service more requests, which allows it to scale up better.

There are 2 ways in which this can be implemented:

1. The load balancer receives all the HTTP requests and then forwards it to the app tier, whose multiple replicas are present. Each app tier replica has its own database replica. If the database is updated frequently, it also needs to be synchronized frequently. Thus, this is mostly used in databases that are read heavy and are infrequently updated. This supports distributed replicas.

2. The load balancer receives all the HTTP requests and then forwards it to the app tier, but all the multiple app replicas have a common database tier, that is they connect to the same component. This is more commonly used by multiple applications. In this case, no data synchronization is required. This works well as long as bottleneck is not I/O. In most cases, the bottleneck is request processing, thus this is not a problem. However, this way does not support having distributed replicas of the database, unlike method 1 above.

There are different ways in which the request can be forwarded to the servers in fig 20.6:

1. Request-based scheduling: Every HTTP request coming in can be potentially sent to any replica. This can be done in cases where there is no state to be saved. To facilitate the storing of a state in request-based scheduling, we can have a common storage where all the replicas can access the state of a client from.

2. Session-based scheduling: A browser first establishes a session with the web-server. Once this is done and maps to a replica, all requests of that server are sent to the same replica. This is beneficial as opposed to request-based scheduling as it helps in caching, helps in keeping the state in a single machine (example: saving the state of a shopping cart).

Irrespective of which way a request is forwarded, the way the load balancer can forward requests to one of the replicas, is either by using HTTP redirect, TCP splicing or TCP handoff. In the context of figure 20.7, the switch (load balancer) needs to send the request to some other server in order to process it.

1. HTTP redirect: The load balancer simply redirects the HTTP request to the web server, which then processes the request and sends the response back.

2. TCP Splicing: The client sends the HTTP request to the switch, which then makes another request to the web server. When the reponse comes back, the load balancer appends this to the first request and sends it back. These are essentially 2 connections that are spliced together.

3. TCP Handoff: Similar to HTTP redirect where we get a TCP connection and we hand it off to another machine. It requires network level changes to handoff using socket connections.

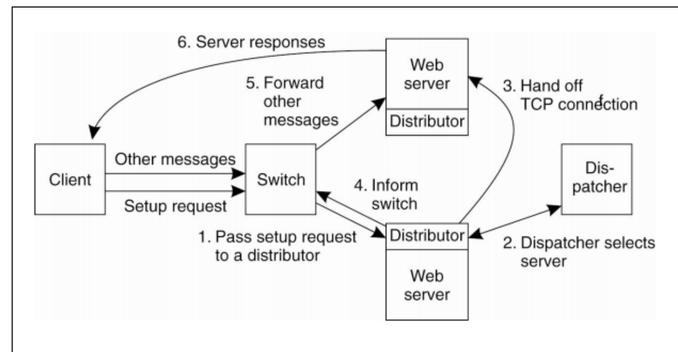

Figure 20.7: Scalable Content-Aware Cluster of Web servers

Question: Can you give some use-cases for these?
Answer: They all do the same thing. For HTTP redirect to work, all the machines would have to have a public IP address so that the client can connect to it. But if we have private IP addresses and only the switch has a public IP, we would not be able to use HTTP redirect, and would instead have to use splicing or TCP handoff.

Question: Will HTTP redirect have session information?
Answer: When any client sends a request to the switch, the switch keeps a table saying that it received a request from that client. It checks if this is a part of an existing session. If so, it takes that machine and does an HTTP redirect to it or sends the request using any of the other mechanisms. Thus, session/state and redirection mechanisms are somewhat orthogonal. However, to an extent, there are schedulers that would do an HTTP redirect and the client can send subsequent requests directly to that server.

Question: Suppose we receive an HTTP request and do some partial processing, and figure out that it needs to be sent to a subsequent process? Can we do the handoff then?
Answer: The multitier architecture does exactly that. We don't have to handoff the request, we can make another request and get its response. How to ask a client to forward all its subsequent requests to another server? This has other mechanisms, but even if some partial processing is done, we can still do an HTTP redirect request, as we haven't sent back a response yet.

Question: Is HTTP redirect a two-step process wherein we query the switch, it sends back an IP and then we send the request to this web server?
Answer: Yes, in HTTP redirect, we get a new URL to effectively talk to. In TCP splicing, we have the request already and make another connection and send it there, basically relaying the request. TCP handoff does something similar.

20.7 Elastic Scaling

It is an interesting technique that can be implemented when we have a clustered web application. Web workloads are time-varying (time-of-day effects, seasons when the workload is high, etc). There are other kinds such as load spikes or flash crowds, wherein the workloads increase suddenly (example: news story breaks). Some may be expected, such as sports events, big sales, and so on. How to deal with these changing workloads?

One approach is to decide the absolute maximum workload that the service will see and put enough servers to be able to handle it. But many-a-times, a lot of the servers would be sitting idle. Also, it is not always possible to predict the traffic. As a result, applications are generally under-provisioned, wherein the workload exceeds the capacity even when there are multiple replicas.

Elastic scaling/auto-scaling: Increase the capacity on the fly with increasing loads. The web server monitors its threshold and adds servers when this threshold is being approached. This can be done programmatically in cloud applications. There are 2 ways:

1. Horizontal scaling: There are multiple replicas and we add or remove those based on the load.
2. Vertical scaling: We don't change the size of the cluster, but change the size of the replicas, by giving them more cores.

This is used widely in modern cloud based applications. When do we scale? Look-ahead and predict workloads (maybe for the next hour) and pre-provision resources ahead of time. This is called proactive scaling. Another is reactive scaling. This is when we don't do any provisioning but monitor the load. For example, we only add new machines when the load reaches 70% or 80%. This may lead to small disruptions due to time taken to start the machines.

20.8 Microservices Architecture

Each application is a collection of smaller services. We take an application tier, which is basically a monolithic application tier and split it into smaller components, each of which is a microservice. This is also an example of a service-oriented architecture. This gives modularity and we can change each microservice independently, without affecting the other microservices and making it easier to manage. Teams can be responsible for one service. These can be independently deployed as well. Each microservice can be clustered and auto-scaled. Example: we can scale up only that microservice which is compute intensive. But this makes the application look more complicated. This is one more way of scaling web applications. The figure 20.8 shows 3 ways of scaling. The x-axis is horizontal scaling. z-axis is called data partitioning: we partition the data instead of replicating it. If we don't want to replicate our data but it becomes a bottleneck, one way is to split the data into parts and put it onto different machines. This is sharding or partitioning. The y-axis shows functional decompostion, wherein we take different microservices and scale them independently. We can use any combination of these, typically all the 3 are used.

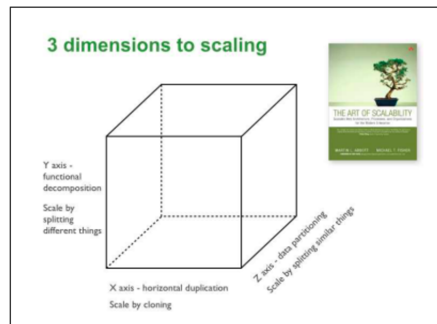

Figure 20.8: Scalability cube which shows 3 ways of scaling

20.9 Web Documents

Most web browsers get back content, mostly an HTML page with embedded content or objects. The figure 20.9 shows the types of objects. The simplest is the text object. The content is encoded using MIME. Web browsers extract it, parse it and render.

Web Documents

Type	Subtype	Description
Text	Plain	Unformatted text
	HTML	Text including HTML markup commands
	XML	Text including XML markup commands
Image	GIF	Still image in GIF format
	JPEG	Still image in JPEG format
Audio	Basic	Audio, 8-bit PCM sampled at 8000 Hz
	Tone	A specific audible tone
Video	MPEG	Movie in MPEG format
	Pointer	Representation of a pointer device for presentations
Application	Octet-stream	An uninterpreted byte sequence
	Postscript	A printable document in Postscript
	PDF	A printable document in PDF
Multipart	Mixed	Independent parts in the specified order
	Parallel	Parts must be viewed simultaneously

Figure 20.9: Web Documents

20.10 HTTP Connections

Figure 20.10 shows the original HTTP 1.0, where browser sets up a new TCP connection to the server every time we make a HTTP request. Each entity or object we get back has its own connection. Once we get a response, we tear down the connection. That is, it is non-persistent. Making new connection to the same server every time is wasteful. A variant, version 1.1 20.11 is more efficient as it creates persistent connections. We send next HTTP requests over the same connection. The browsers does not close the connection immediately, in anticipation of further requests. However, these are sequential. We need to get he response to the previous request before sending the new request. This can be slow and many browsers don't use this mechanism. For multiple simultaneous downloads, we set up multiple connections instead. Thus, HTTP1.1 did not solve its goal of saving connection overhead due to sequential requests.

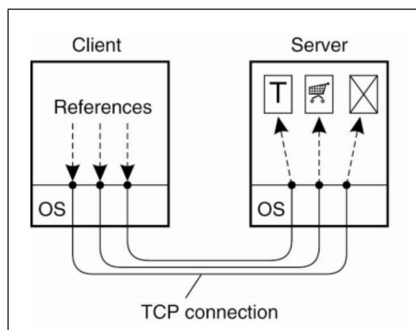

Figure 20.10: HTTP 1.0: Using Non-Persistent Connections.

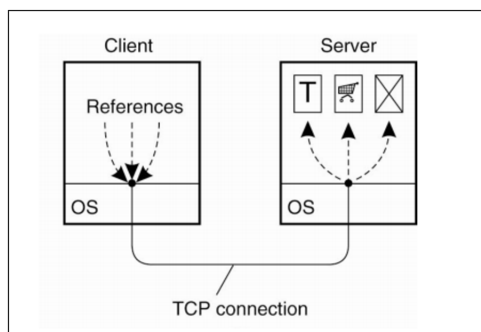

Figure 20.11: HTTP 1.1: Using Persistent Connections.

20.10.1 HTTP Methods

HTTP protocol has five methods:

1. The simplest among them is the 'GET' command. It takes a URL and simply fetches what the URL is pointing to, which can be an HTML page, an image, etc..

2. While 'PUT' command is used to store a document on the server.

3. 'POST' allows us to add data to the document. Whenever we submit webforms, it is a 'POST' request.

4. 'DELETE' can be used to delete a document, but most web browsers do not support it.

5. 'HEAD' gets the header of the document. It is typically used for caching.

HTTP 2.0 is designed to address the message latency problem. It allows us to have binary headers. We can compress headers and messages, making the message smaller and thus faster. It allows concurrent connections (persistent but with concurrency), thus we do not have to wait for responses for the previous requests. This is done using the concept of streams. Each stream is one request and one response. To send a request, we send it using a new stream, thus when the response comes, we know which request it is intended for. This helps in speeding up the connection significantly. Both the browser and the server have to support HTTP 2.0. It is not backward compatible.

20.11 Web Services Fundamentals

Webservices are ways in which we can write applications and use RPCs between these applications or between a client and a server. The term webservice has a specific connotation where we use a certain type of interface description language, a certain protocol for SOAP for us to send RPCs. We use an interface definition language called webservice definition language (WSDL), a compiler that generates stubs for the client and the server, the protocol SOAP (like HTML but is XML based) used for communication. This can be seen in fig 20.12.

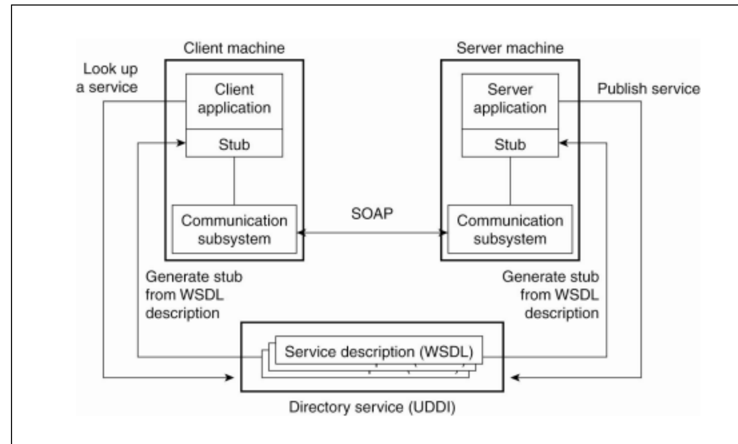

Figure 20.12: Web service

SOAP, Simple Object Access Protocol, was used to make RPC requests over HTTP. Fig 20.13 is an example of making an RPC request over SOAP. The entire RPC request is sent as a XML document. The server after receiving the document, parses it, perform required operation and sends back the response as another XML document. Figure 20.13 shows one such XML request document. Here the client is calling 'alert' method and passing the string 'Pick up Mary at school at 2pm' as an argument to that method.

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
    <env:Header>
      <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
        <n:priority>1</n:priority>
        <n:expires>2001-06-22T14:00:00-05:00</n:expires>
      </n:alertcontrol>
    </env:Header>
    <env:Body>
      <m:alert xmlns:m="http://example.org/alert">
        <m:msg>Pick up Mary at school at 2pm</m:msg>
      </m:alert>
    </env:Body>
</env:Envelope>
```

Figure 20.13: Example of XML-based SOAP message

Question: While communicating between microservices, do we use HTTP 2.0 or HTTP 1.0? **Answer:** To some degree, it depends on what the HTTP library does. The code should not change for this. It should be HTTP 1.0 as HTTP 2.0 is mostly used in browsers.

Question: If we use HTTP 2.0 library and make a new connection, it should not create a new one, right? **Answer:** If we use HTTP 2.0 library and make a new connection, internally it should check that there is already a connection and should not create a new one. Another way is we don't call it repeatedly to create, but just send the request over the existing socket.

20.12 Restful Web Services

As we can see, for calling single method with one argument, we have to send such a long XML file. SOAP did not perform well because of this overhead. As a result, SOAP evolved into Restful architecture. Restful architecture makes RPC request over HTTP. HTTP was already popular than something like SOAP, so it was chosen as a way to make RPC requests. In case of Restful architectures, the communication is light weight and assumes one-to-one communication between client and server.

In Resfult webservices, we use:

1. GET: to read something
2. POST: to create, update or delete something
3. PUT: to create or update something
4. DELETE: to delete something

The example in 20.14, a GET request is made and the response is received. The response sends an XML packet in this case, but could also be JSON, any format can be used. It is much more compact than using SOAP.

```
GET /StockPrice/IBM HTTP/1.1
Host: example.org
Accept: text/xml
Accept-Charset: utf-8
```

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<s:Quote xmlns:s="http://example.org/stock-service">
     <s:TickerSymbol>IBM</s:TickerSymbol>
     <s:StockPrice>45.25</s:StockPrice>
</s:Quote>
```

Figure 20.14: Example of a Restful web service

20.13 SOAP VS RESTful WS

1. SOAP application can be written in various languages and can run on different OS or platforms and is also transport agnostic (XML, TCP, etc. cab be used). It need not use HTTP. Whereas, RESTful services only support HTTP. Restful services, however, can use any language to write this, such as Python, Java, and so on.

2. SOAP is for general purpose distributed systems. We can have all kinds of applications making calls to each other. Restful webservices have a client and a server and is point to point. We have pairwise interactions, just like traditional RPC.

3. SOAP has a wide set of standards (telling us how to write, compile, etc/), whereas RESTful services does not have any pre-defined standards. They have general guidelines, but no one particular way to write it.

4. SOAP is very heavyweight compared to REST.

5. Rest has less of a learning curve compared to SOAP.

20.14 Web Proxy Caching

One mechanism to use proxies is to use for web caching. Typically, we take a URL and send it to the server that answers to that URL. Let's say there is a proxy server sitting near the client, which maintains a cache. All requests are sent to the proxy server, which looks at the cache. If data is found, we immediately send back the response. This helps in faster responses for the client if the proxy is near the client and reduced loads for the server. If we cached a webpage and the webpage changed on the server, the proxy may serve stale content. Thus, cache consistency is important.

Suppose if we are looking for some webpage but it is not there on the proxy, this is called a cache miss. There are 2 things to do. One is to and get it from the server, put it in the cache and send the response. Figure 20.15 shows another method to deal with this, called "Cooperative Caching". Along with communicating with the server, these proxy caches can also communicate with each other. A proxy reaches out to the near by proxies to get the data. If the data is present on nearby proxies, it is faster than reaching out to the server. The client sees the union of all the stored caches.

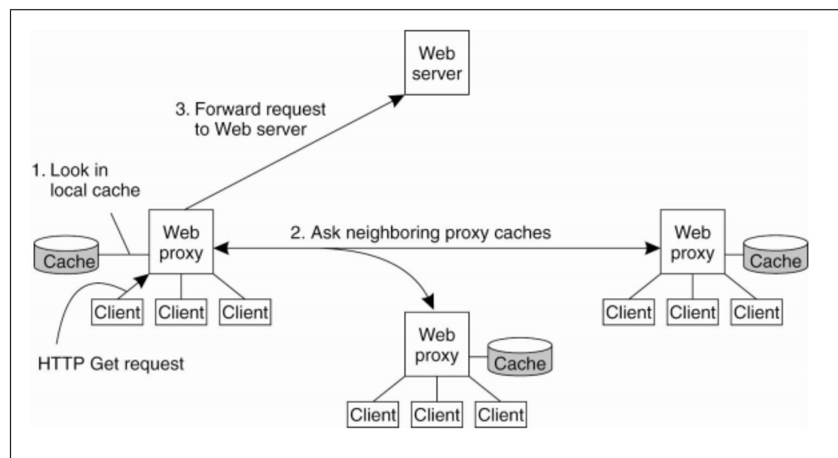

Figure 20.15: Cooperative Caching

20.15 Web Caching

It is also import to deal with consistency. Web pages tend to change with time. When a browser fetches a page from the server, we are guaranteed that the returned page is the most recent version. While using proxies, we need to ensure the consistency of cache web pages. The popularity and update frequency can be different across web pages. We need to consider both these issues for maintaining consistency. That is, if it is a cache hit, how will the proxy know it is the updated data?

1. Pull based approach: We poll periodically and use a conditional GET to ask the server if the cached data has changed
2. Push based approach The web server tracks each proxy and the pages cached there. If there has been an update, it sends a push to invalidate the data.