CMPSCI 677 Distributed and Operating Systems **Spring 2022**

Lecture 2: January 31

Lecturer: ***Prashant Shenoy*** *Scribe:* ***Steven (Jiaxun) Tang (2022), Roy Chan (2019), Phuthipong Bovornkeeratiroj (2018)***

Note: Please make sure that Gradescope is working for you.

Note: There is a Career Fair on February 24.

Reminder: No using laptops or phones during class.

2.1 Architectural Styles

Most distributed systems can be described by one of the architectures discussed in this lecture. It is important to understand the differences between them so that we can decide on the architecture before implementing a new system.

2.1.1 Layered Architectures

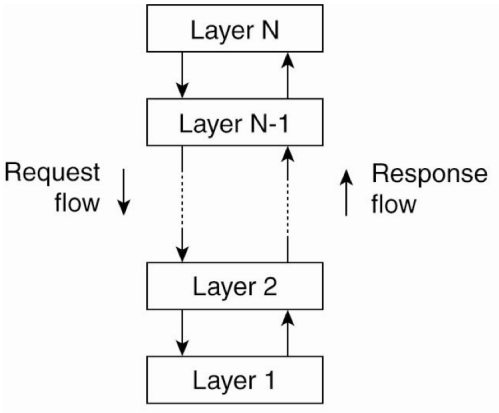

Figure 2.2: Layered Design

A layered architecture looks like a stack, as seen in the figure above. The system is partitioned into a sequence of layers and each layer can communicate to the layer above or below. For example, layer i can communicate with layer $i+1$ and layer $i-1$ but not the others (e.g. layer $i+2$). This is the main restriction of a layered design. The layered architecture is especially common in web applications where this architecture is divided across the client and the server. Common instances of these systems are multitiered architectures and network stack.

2.1.2 Object-Based Style

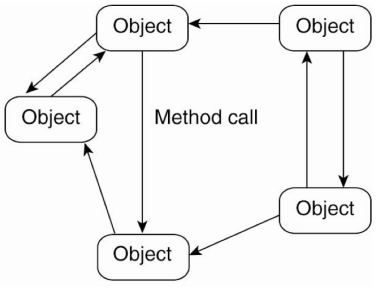

Figure 2.3: Object-based Style

In this architecture, each component corresponds to an object. Unlike in standard OOP programming, objects can be distributed across multiple machines. As shown in the figure above, the system can have many objects. Each object have has its own states and exposes its own interface which other objects can use. All objects can communicate with any other object without restriction, making this a "generalized" version of the layered design. Components interact with each other via remote procedure calls. We will discuss RPC in Lecture 3.

2.1.3 Event-Based Architecture

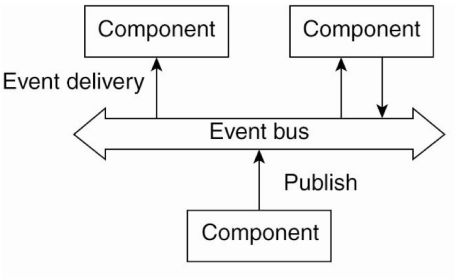

Figure 2.4: Event-based Architecture

An event-based architecture has many components that communicate using a publisher-subscriber (pub-sub) model via an event bus instead of through direct communication. In this architecture, a component that sends an event to the event bus is a publisher, and a component that subscribes to certain types of events on the event bus is a subscriber. Each component will work asynchronously. After a component sends information by publishing an event, the event bus then checks for subscriptions matching the recipient information enclosed in the newly published event. If one or more matching subscriptions is found, the event bus will deliver the data to the appropriate component(s). There are many kinds of event busesk, e.g., memory-based or disk-based.

2.1.4 Shared Data Space

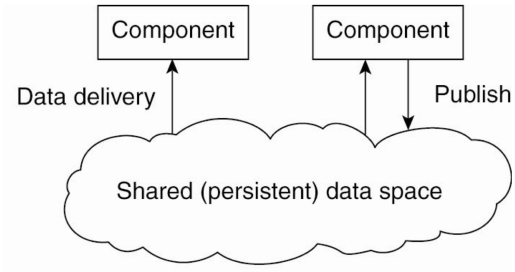

Figure 2.5: Shared Style

The shared data space architecture has a shared data space which is like a physical bulletin board. A component posts information and some component may come along later and retrieve the information. Unlike in the event-based architecture, data posted in the shared data space have no specific information about the recipient. Therefore, posted data can be in the shared data space for a while until some component actively retrieve this data. From this sense, the components in the data-space architecture are loosely coupled in space and time. Notice that the data that is published is not addressed to anyone in particular, and that the data may not be recieved in real-time.

Question (Student): Does the shared data space architecture require a centralized server?

Answer (Instructor): The shared data space model is like a distributed database. It does not have to be centralized and can be done in a distributed manner.

2.1.5 Resource-Oriented Architecture (ROA)

A resource-oriented architecture exposes resources for clients to interact with. Resources have names and related operations. Representational State Transfer (REST) is a common implementation of this architecture. It has a standard naming scheme in which all services offer the same interface (GET/PUT/POST/DELETE). No client state is kept, which means each request is logically decoupled. Since users often interact with the resources of a web service, exposing application as resources make it easy to implement descriptive APIs. For example, if you want to query/create/delete/update an object in the Amazon Object Storage Service S3, you just need to send GET/PUT/DELETE/POST requests to `https://{{BUCKET_NAME}}.s3.aws.com/{{OBJECT_NAME}}`.

Question (Student): REST services are stateless because all the information is in the URL. What about authentication? Would you maintain states for that?

Answer (Instructor): It all depends on how you design the web service. Once authenticated, it can provide a token. You can do authentication in a stateless manner.

Question (Student): Wouldn't there still be a state sometimes?

Answer (Instructor): The service will typically be stateless.

2.1.6 Service-Oriented Architecture

A service-oriented architecture exposes components as services. Each component provides a service. Services communicate with each other to implement an application. Micro-services are one modern implementation

of a service-oriented architecture.

Question (Student): What is the difference between SOA and ROA?

Answer (Instructor): SOA exposes components as services and ROA exposes components as resources. ROA requires services to connect via HTTP, while SOA doesn't enforce the protocol used. ROA is stateless and SOA can be stateful. ROA is also newer and is better for using HTTP.

Answer (Instructor): ROA is stateless and SOA can be stateful. ROA is also newer and is better for using HTTP.

The following are comparisons between OOA, ROA, and SOA.

Attribute	Object-oriented	Resource-oriented	Service-oriented
Granularity	Object instances	Resource instances	Service instances
Main Focus	Marshalling parameter values	Request addressing (usually URLs)	Creation of request payloads
Addressing / Request routing	Routed to unique object instance	Unique address per resource	One endpoint address per service
Are replies cacheable?	No	Yes	No
Application interface	Specific to this object / class – description is middleware specific (e.g. IDL)	Generic to the request mechanism (e.g. HTTP verbs)	Specific to this service – description is protocol specific (e.g. WSDL)
Payload / data format description	Yes – usually middleware specific (e.g. IDL)	No – nothing directly linked to address / URL	Yes – part of service description (e.g. XML Schema in WSDL)

Courtesy: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/3-arch-styles.pdf

Figure 2.6: OOA vs. ROA vs. SOA

Question (Student): Can a system use hybrid architectures?

Answer (Instructor): Yes. For example, an application that uses micro-services can also implement RESTful API. You can often choose a base architecture and compose other architectures together according to your needs.

2.2 Client-Server Architecture

This is the most popular architecture. The client sends requests to the server, and then the server sends a response back to the client. Remember that this does not necessarily refer to the hardware. The terms "client" and "server" refer instead to the piece of software that requests the service or provides the service. After the client sends a request, it waits while the server processes the request. In the figure below, you can see the respective parties waiting when there is a dotted line.

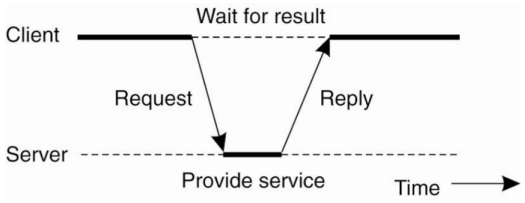

Figure 2.6: Client-Server Architecture

Developers need to make design choices about which service should be put into which layer. Let us look at an example to see how we would implement this.

2.2.1 Search Engine Example

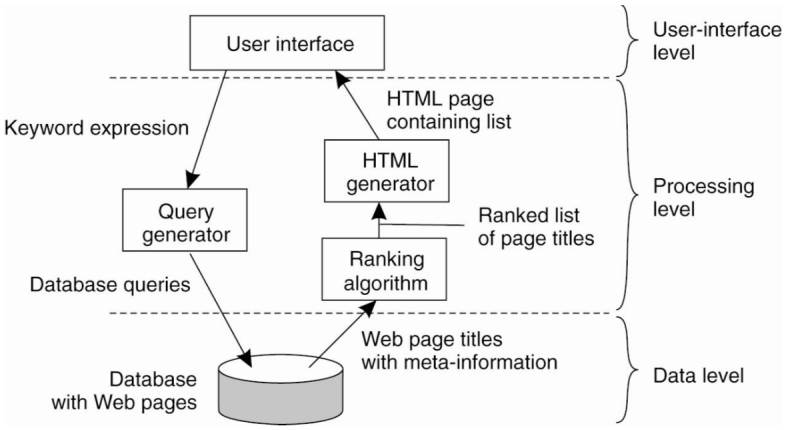

Figure 2.7: Search Engine Example

Take Google search as an example. When you type something into the search box, you are interacting with the UI level of Google search. Then, the UI will send your input to a query generator at the processing level. The query generator translates your query expression into database queries and accesses the database located at the data level. A ranking algorithm in the processing level takes the query results, ranks them, and passes the result to the HTML generator at the same level. The generated page is then sent back to the UI layer and will be rendered by the browser as a webpage. The important part is understanding the tiers and how they interact with each other in a distributed application. Other details like indexes and crawlers are not the components we are considering here.

2.2.2 Multitiered Architectures

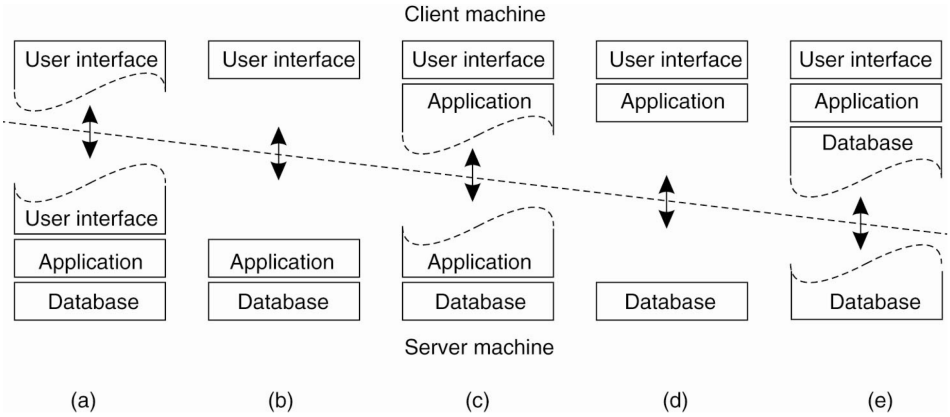

Figure 2.8: Client Server Choices

We see various "splits" of the 3 layers between client and server represented by the dotted line. The layer(s) above the line are on the client, and the layer(s) below the line are on the server. As you can see, there are many choices in how you split the implementation.

A typical implementation of (a) is a traditional browser-based application (e.g. SPIRE). The webpage is constructed from the server-side and rendered in the browser. A typical implementation of (b) is single-paged web application. The server does not render pages, but only provides APIs for data retrieval. The browser will send AJAX requests to call those APIs. A typical implementation of (c) is a smartphone app, where the application's backend is usually split between the device and the server. Desktop applications usually follow (d) where only the database is on the server, and the client is just accessing data. A smartphone app or a whole app that exists on a client also follows this architecture. Lastly, (e) improves on (d). Data is cached or stored locally. For example, Google's offline mail caches a small subset of the user's email locally. The choice of which architecture to use depends on many factors, e.g., what you want to do, how much resources the client has, etc.

2.2.3 Three-tier Web Application

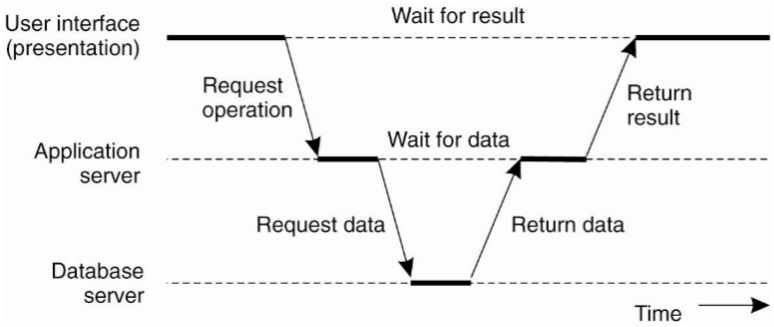

Figure 2.9: Three-tier Web Application

The three-tier web application architecture is a very popular architecture choice. The client's browser sends an HTTP request to an HTTP server (e.g. apache). The HTTP server then sends the request to the app

server (e.g. a Python backend) for processing in which it may create a query to the database server. The database returns data to the app server that sends the results to the HTTP server which then forwards it to the browser. The sequential nature of this architecture is a type of layer architecture seen earlier in the search engine example.

These tiered architectures can use more or fewer than 3 layers depending on their setup. Modern web applications will take the Application tier and split it into multiple tiers. A very common architecture for web apps uses HTTP for the user, PHP or J2EE for the app server, and then a database for the bottom tier. The divide between user and server is not set in stone as we saw in the previous section.

2.2.4 Edge-Server Systems

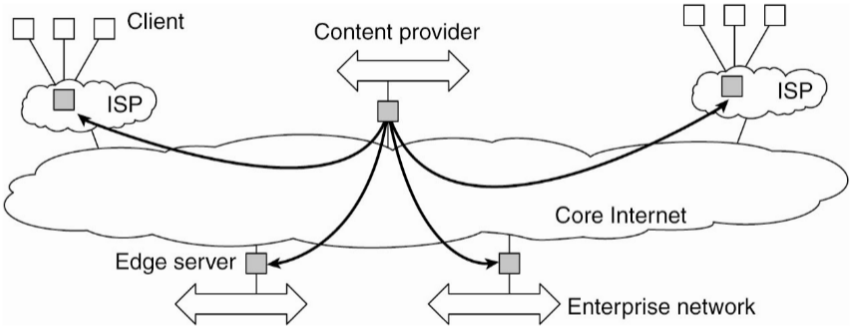

Figure 2.10: Edge Server

Unlike traditional client-server architecture, edge server systems implement a client-proxy-server architecture. As the name suggests, there is an extra component in between. The proxy (labeled as the edge server in the figure above) sees if it can process the client's request without having to go to the server (i.e. the Content provider). If not, the proxy forwards the request to the real server. The advantage of this approach is that the main server load is reduced, and data is moved to servers closer to the user so that the access latency will be greatly reduced. Many other proxy services can be provided in addition to caching. Edge computing goes one step further than simply providing a data cacheby allowing code execution in the edge server.

2.3 Decentralized Architectures (Module 3)

Decentralized architectures are also known as peer-to-peer (P2P) systems. Unlike the client-server architecture, each node (peer) can be a client, server, or both with all nodes being mostly equal. That is, we are removing the distinction between client and server. P2P systems can also come be structured or unstructured systems. A peer can provide services and request services. Peers can also come and go at any time, unlike a server which must be there all the time. We will introduce a structured peer-to-peer system named "Chord" as an example.

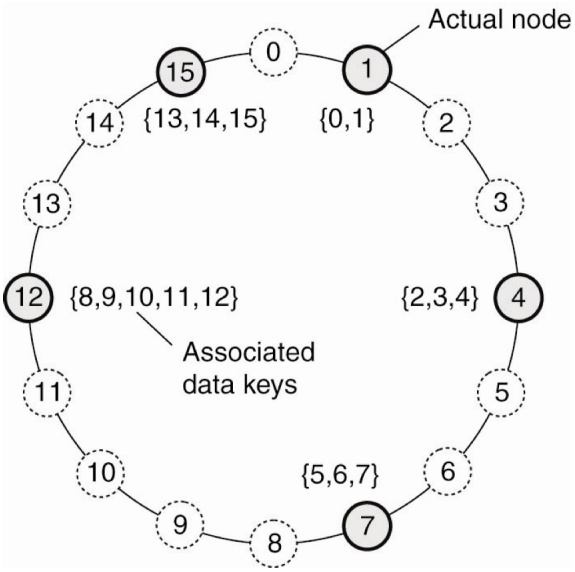

Figure 2.11: Chord Structure

The Chord system maintains a hash function to associate data nodes with an integer key. In this figure, there are $n = 16$ keys in the system. The darker circles are peers that already joined the Chord. Node 1 is responsible for storing data $\{0, 1\}$, node 4 is responsible for storing node $\{2, 3, 4\}$ When a node joins, it picks an ID that is a key and is unfilled from 1 to n and then stores keys from the previous node to itself. How one chooses the key for a joining node can be random or structured. In our current case, when n_7 joined, it became responsible for storing [7, 6, 5]. When a node leaves, the chord structure assigns the leaving node's keys to the next node above it. If n_7 were to leave, n_{12} would then be responsible for $\{12, 11, 10, 9, 8, 7, 6, 5\}$. As one can see, joins and leaves are symmetric. Replication or redundancy is used so that when the node leaves, the system still works.

Given a key in a request, the system has to figure out what node has that key. This can cause request routing, in which the system will hop around nodes until the key has been found. Fortunately, the search is actually fast, with a provided key, the system has to look up the value in the distributed hash table. The hash table is provided by the distributed hash table (DHT) algorithm. P2P architectures are not as reliable as client-server architectures, as peers can join and leave the network without advance notice. A technique called "consistent hashing" ensures the DHT is fault tolerant.

More details about Chord can be found here:
`https://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf`

Question (Student): What is the value that is stored in the hash table?

Answer (Instructor): It is a hash table so anything that can be in it. Usually it can store a file, i.e., the service is a file lookup. But it is like asking, "What can you store in a database?" Whatever you want!

Question (Student): Does a P2P architecture imply an ad-hoc network? That is, do nodes just come and go?

Answer (Instructor): It is designed to ensure high reliability. Unlike in a client-server architecture where the server is assumed to be reliable, nodes may not be reliable so the system is built to handle nodes joining and leaving.

2.3.1 Content Addressable Network (CAN)

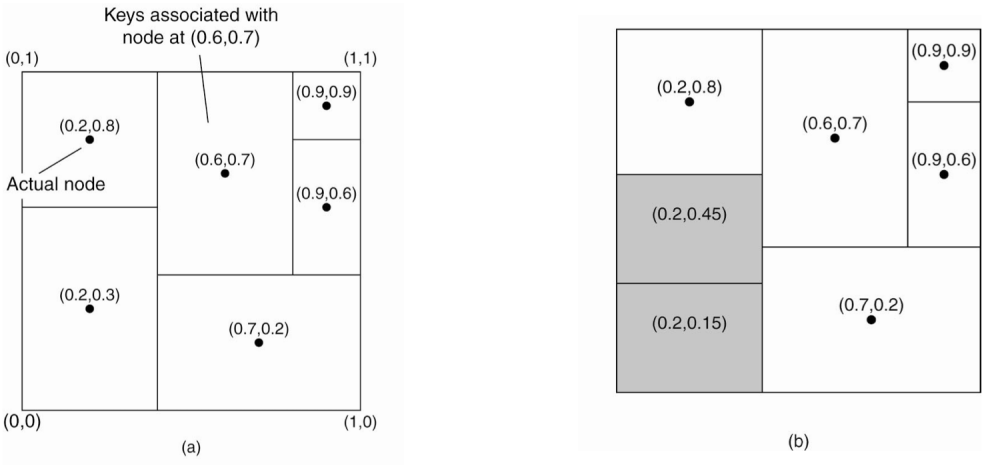

Figure 2.12: CAN Structure, with (b) showing a join procedure

Content Addressable Network (CAN) is another P2P system. As opposed to Chord, however, CANs are generalized versions of Chord, i.e., they use a d-dimensional coordinate system. To make illustrations easier, we will set $d = 2$ for the rest of this section. For example, we can have a tuple containing a file name and a file type which would require a two-part key for the two-part attribute. Here, each piece of content in a CAN has 2 identities: $< id.x, id.y >$ or <file name, file type>. For example, two files named "Foo" may have different file types such as .jpg and .txt.

In the figure above, each dot is a node, meaning that each node is responsible for a rectangular partition of the coordinate space. The user can have a more fine-grained query in this structure. The x-axis and y-axis are showing normalized values of the keys from 0 to 1. If a node joins, it chooses a random (x, y) coordinate and splits the box (i.e. a specific coordinate space) that it is in with the existing node. A node leaving is more difficult, as the merging of 2 rectangles is not always a rectangle. If a node leaves, the system must partition that rectangle to merge it with other already present rectangles. Consistent hashing again ensures the correct handling of the hash when nodes exit.

Note: Remember that the specific example here shows 2 dimensions, but CAN could have any d-dimensional coordinate system.

Note: In Chord, one can also represent the <file name, file type> attribute, but this would require concatenating the 2 keys into one.

Question (Student): Does this require a full replication of content on all peers?

Answer (Instructor): No, because that assumes all nodes will leave at the same time. Say, for example, each node replicated at 3 other nodes. This provides the assumption that if 3 nodes leave, we're still ok.

2.3.2 Unstructured P2P Systems

Rather than adhering to some topological protocol such as a ring or a tree, unstructured topologies are defined by randomized algorithms, i.e., the network topology grows organically and arbitrarily. Each node picks a random ID and then picks a random set of nodes to be neighbors with. The number of nodes is based on the choice of degree. If $k = 2$, it means the new node will randomly link to 2 existed nodes and

establish logical connections. When a node leaves, the connections are severed and any remaining nodes can establish new links to offset the lost connections.

Without structure, certain systems can become more complicated. For example, a hash table key lookup may require a brute force search. This floods the network, and the response also has to go back the way it came through the network. We observe that the choice of degree impacts network dynamics (overhead of broadcast, etc.). The unstructured notion of such P2P systems framed early systems, but newer systems have more structure in order to reduce overhead.

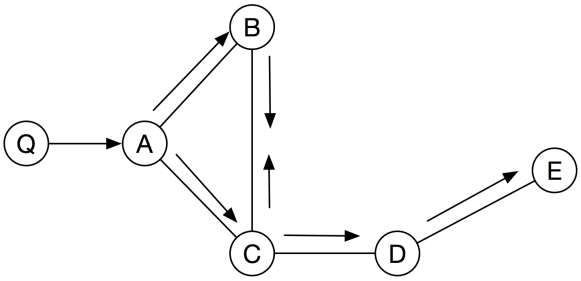

Figure 2.13: Search in Unstructured P2P System

From the figure above, we see that search in an unstructured P2P system is done by propagating through the graph as seen in the above figure. Here, a query (Q) is passed to node A, which is then propagated through the network as each node queries its neighbors. Eventually, the signal is backpropagated to the sender. This can easily flood the system as mentioned above, so one can create a hop count limit to reduce unnecessary traffic. Each time the query is passed to a neighbor, the hop count is decremented. Upon reaching 0, the node will simply return not found.

2.3.3 SuperPeers

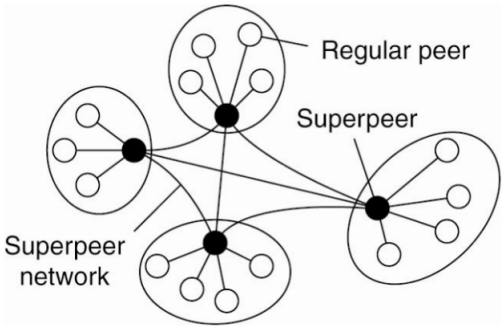

Figure 2.15: Graph with SuperPeer Structure

A small modification to the completely unstructured P2P system allows for much more efficient communication and reduces overhead. The P2P graph is partitioned into clusters, where one peer, designated to be the superpeer, within each cluster can communicate with other peers outside of the cluster. These superpeers are dynamically elected within each group and should have additional resources to facilitate the increased communication demand.

The restricted communication reduces unneeded calls to neighbors and prevents the huge amount of broadcast

traffic found in the completely unstructured P2P system. The number of messages should be lower. However, there may still be a lot of traffic still flooding the network albeit only going through superpeers.

An early versions of Skype was a good example of how superpeers work. It tracked where users were and if they were logged in from a specific cluster. It was a P2P system, but Skype has now moved to a client-server architecture instead.

Question (Student): What are some more examples of superpeers?

Answer (Instructor): BitTorrent and P2P backup systems. However, whenever an application is very important, they may not use P2P since P2P assumes that people are donating resources to make the system work.

Question (Student): Are node link connections static or dynamic?

Answer (Instructor): We can't assume that neighbors will stay up. The topology is constantly changing so we must assume dynamic connections and that links with new neighbors will be made.

2.3.4 Collaborative Distributed Systems

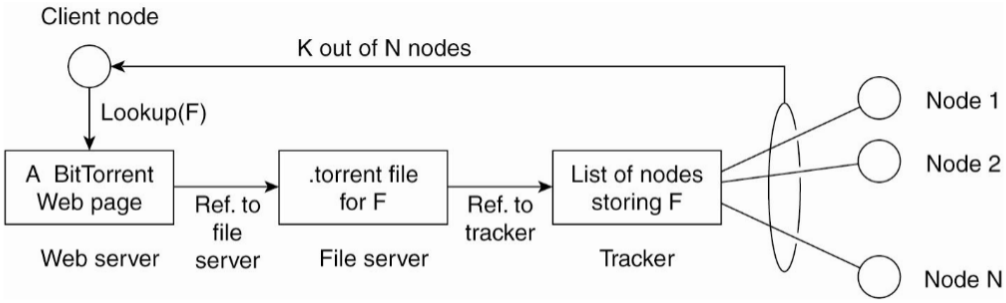

Figure 2.16: BitTorrent, an example of a collaborative distributed system

In a collaborative distributed system, files are split into chunks and spread across peers. A client can request these chunks and piece them together. This system allows parallel file download sources from multiple connections, which is faster than a sequential file download from a singular connection. A node can control how parallel it wishes to be, i.e., how many nodes or peers it connects to.

A system like BitTorrent can also take into account an altruism ratio, and slow down a node based on the ratio. If a node is just downloading without also uploading chunks in its possession, or more generally, provide services to other peers, then the system may reduce the download speed of the node. This incentivizes nodes to participate in and contribute to the network instead of freeloading so that they can get good performance.

Two key components are involved in a torrent system: the tracker and the torrent file. The tracker is an index that monitors which nodes have which chunks. The torrent file points to the tracker and can be posted on a web server. In short, the torrent file gets a client node to the tracker which shows which peers it needs, and then the client node can directly connect to those peers based on the configurable setting of how many peers it wants to connect to at one time.

Question (Student): Does the tracker get updated?

Answer (Instructor): As long as a user is connected to it, the tracker knows who has what content.

Question (Student): How do nodes agree on how a file is split?

Answer (Instructor): The file is split how you want. This is a configurable parameter in the system.

2.3.5 Autonomic Distributed Systems

An autonomic distributed system can monitor itself and take action autonomously when needed. Such systems can perform actions based on the system performance metrics, system health metrics, etc. We will not dive too deep into this topic, but knowing the concept helps.

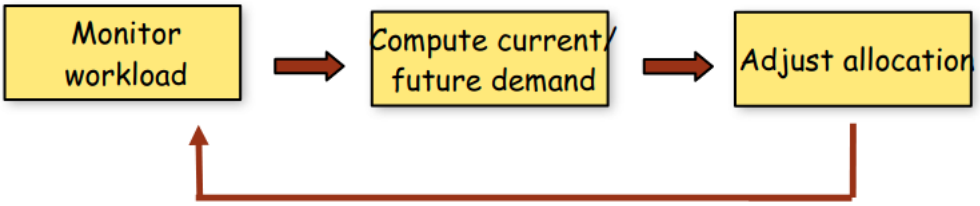

Figure 2.17: Automatic capacity provisioning

This is an illustration of how you might implement such a system. You can monitor the current workload, predict future demand, and if the system thinks the current resource is not enough, deploy more nodes. If the system thinks the resource is not enough, then the number of nodes could be reduced. This technique is also called elastic scaling. The workload prediction part can involve many techniques. For example, we could use feedback and control theory to design a self-managing controller. Machine learning techniques such as reinforcement learning can also be used.