CMPSCI 677 Distributed and Operating Systems **Spring 2019**

Lecture 2: January 28

Lecturer: ***Prashant Shenoy*** *Scribe:* ***Roy Chan (2019), Phuthipong Bovornkeeratiroj (2018)***

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

Note: There is a Career Fair on February 12.

Reminder: No using laptops or phones during class.

2.1 Lecture 1 (cont.)

2.1.1 Multicomputer Operating Systems

Here, a distributed operating system provides abstraction on a single logical machine for the user to see even though there are multiple machines in a cluster. The OS handles scheduling internally, and the user logically sees a unified system despite the distributed machines.

2.1.2 Network Operating Systems

A network OS differs from a distributed OS in that the presence of multiple machines is visible to the user. This OS allows the user to communicate over a network. Thus, all general-purpose OS's today are a type of network OS. Distributed applications can still be run, but they must determine which components they need, which machines those components are on, and explicitly communicate with those components. An example of this is a file server that is visible to and can be accessed by users (clients) that are running network OS's. An example of this is NFS (Network File System).

Question (Student): Is it similar to a workstation?

Answer (Instructor): The current discussion is about the architecture of operating systems and not client-server models.

2.1.3 Middleware-Based Systems

These systems add a software layer on top of a network OS and shows abstractions similar to a distributed OS. Users will see the system as if it were a distributed OS.

2.1.4 Comparison of Systems

Item	**Distributed OS**		**Network OS**	**Middleware-based OS**
	Multiproc.	**Multicomp.**		
Degree of transparency	Very High	High	Low	High
Same OS on all nodes	Yes	Yes	No	No
Number of copies of OS	1	N	N	N
Basis for communication	Shared memory	Messages	Files	Model specific
Resource management	Global, central	Global, distributed	Per node	Per node
Scalability	No	Moderately	Yes	Varies
Openness	Depends on OS	Depends on OS	Open	Open

Figure 2.1: Comparison of Systems

Here we can see the various types of OS's compared by their attributes and design.

2.2 Lecture 2

Distributed systems fall into one of the architectures in this lecture. It is important to understand the differences among them so that we can decide on the architecture before implementing a new system.

2.3 Architectural Styles (Module 1)

2.3.1 Layered Architectures

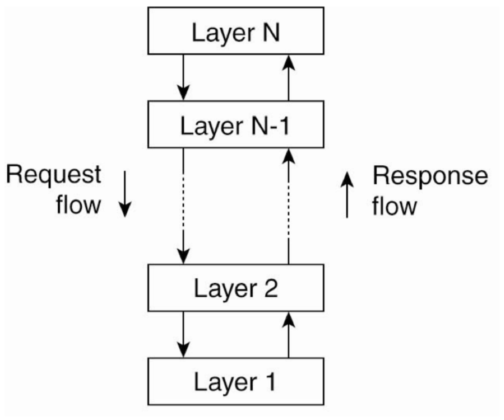

Figure 2.2: Layered Design

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 only an above or below layer. 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.

A common instance of these systems are Multitiered Architectures.

2.3.2 Object-Based Style

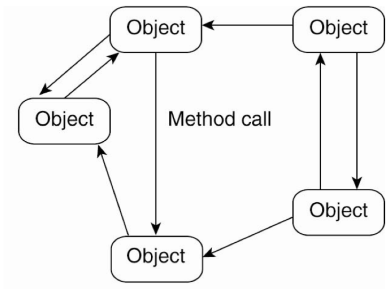

Figure 2.3: Object-based Style

The concept of this architecture is similar to object oriented programming (OOP). The distributed application is split among components i.e. objects. The objects can be distributed across multiple machines. As shown in the figure above, the system can have many objects and each object 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.

2.3.3 Event-Based Architecture

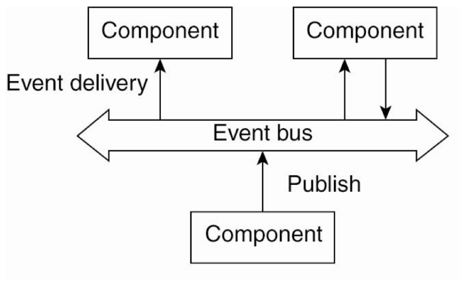

Figure 2.4: Event-based Architecture

An event-based architecture has many components that communicate using a publisher-subscriber (pub-sub) model via an event bus instead of 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 data, the event bus then checks for matches of subscriptions from other components to the newly published data event. If so, the event bus will deliver the data to the appropriate component.

There are many kinds of event buses e.g. memory-based or disk-based.

2.3.4 Shared Data Space

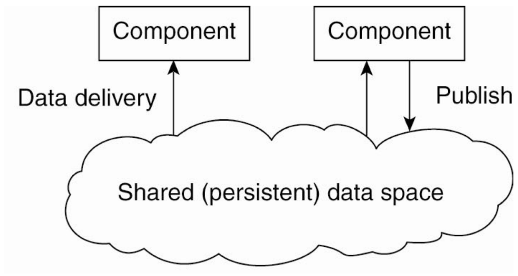

Figure 2.5: Shared Style

Also referred to as "blackboard architecture," this architecture is a loose form of pub-sub, but it decouples in time and space. It can be viewed as a form of a "bulletin board." The publisher posts a note to the shared space, and then a subscriber will look for notes of interest. The data that is posted can be in the shared data space for a while so the subscriber may not get real-time data. This architecture can be either push or pull model.

Decoupled in space means the data that is published is not addressed to anyone in particular.

Decoupled in time means the data that a component can get may not be real-time data.

Question (Student): How is shared data space different from the event-based model?

Answer (Instructor): In the shared data space architecture, data could stay in the shared space for a very long time due to the decoupling. The event-based model is similar to a mailing list in which the data is sent along to subscribers. The shared data space is like a physical bulletin board; a component just posts information, and some component may come along later.

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.

2.3.5 Resource-oriented Architecture

Representational State Transfer (REST) is a popular architecture for web service (API) allowing for specific access. It has a standard naming scheme which all services offer same interface (4 HTTP operation which are GET/PUT/POST/DELETE). The application is split into resources, and components ask for resources via an interface. This architechture is also fully descriptive i.e. all the parameters for operation are in the API.

An example is a link to Amazon Web Server S3 which is a simple storage and is used for web services. The web service may return in JSON format which is a key-value component. XML is another popular format in the past. Another example of a RESTful API is Google Maps which allows specific access for maps of different locations.

We will see this concept in a future lab.

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

Another architecture is service based architecture (SOA), which is popular in industry. Here we are talking about resource based architecture (ROA). In SOA, the application is computing the components; each module provides a service, communicating among them to implement an application. On the other hand, ROA focuses on data and not the computation i.e. the focus is on what resources are accessed. In some scenarios, if one views a service as a resource, then the distiction decreases.

Question (Student): Are clusters SOA?

Answer (Instructor): Here we are talking about the application.

Question (Student): What are the advantages of transitioning from SOA to ROA?

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

2.4 Client-Server Architecture (Module 2)

This is the most popular architecture. The client sends request to server, and then server sends a response back to 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.

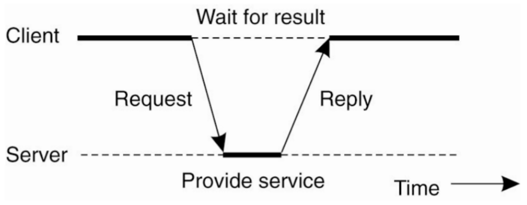

Figure 2.6: Client-Server Architecture

This is also a multitiered architecture. The application is divided into 3 layers: user-interface level, processing level, and data level.

Let us look at an example to see how we would implement this.

2.4.1 Search Engine Example

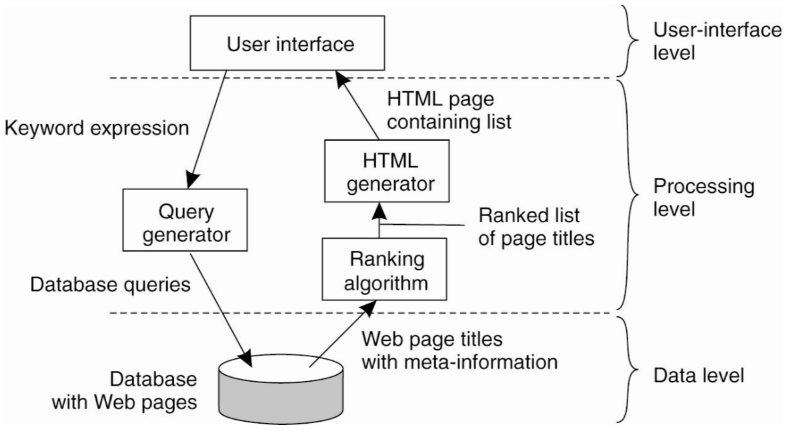

Figure 2.7: Search Engine Example

At the UI level, an example is the Google search box. Going to the Query Generator, we cross the dotted line which represents the transition from client to server in this specific example. At the processing level, the database processes the query. At the data level, the appropriate pages are retrieved and sent back to the processing level where they are ranked and compiled into an HTML page. This is then sent back to the UI level and shown to the user.

The focus is the 3 layers of a distributed application. Other details like indexes and crawlers are not what we're looking at here; the important part is understanding the tiers and how they interact with each other in a distributed application.

2.4.2 Multitiered Architectures

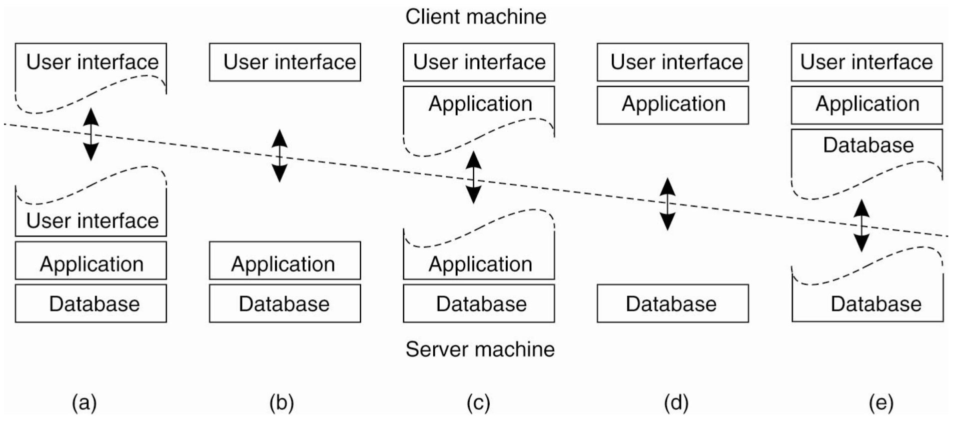

Figure 2.8: Client Server Choices

Usually, the client implements the UI level, and the server handles the other 2 layers. This may not always be the case, however, as you see in the figure above.

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.

Here, (a) represents a browser-based application or a Javascript-enabled website (e.g. Gmail, SPIRE). The UI is split so the browser will perform some processing and the server can perform some work on the interface such as error checking or validating the input.

Question (Student): Can we assume that the client is the user and the server is a system administrator?

Answer (Instructor): We don't really have a system administrator, but we definitely have users accessing on a client.

The architecture in (b) is used by some web browsers e.g. web forms like the Google search bar. Note that if we are also considering autocorrect in the Google search bar, this would be an example of (a) instead since processing would happen in both the browser and the server.

Question (Instructor): What would be some examples of (c)?

Answer (Student): A privacy-preserving ML.

Answer (Instructor): If part of it is on a phone, then yes; not all the data would be sent off to the server.

Question (Instructor): Other examples of (c)?

Answer (Student): Online games.

Answer (Instructor): Yes.

(c) could be an instance of a smartphone app, where the application's backend is usually split between the device and the server. It could also be a game, but the application would have to be on both client and server side.

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 follow 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.j

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.

2.4.3 Three-tier Web Application

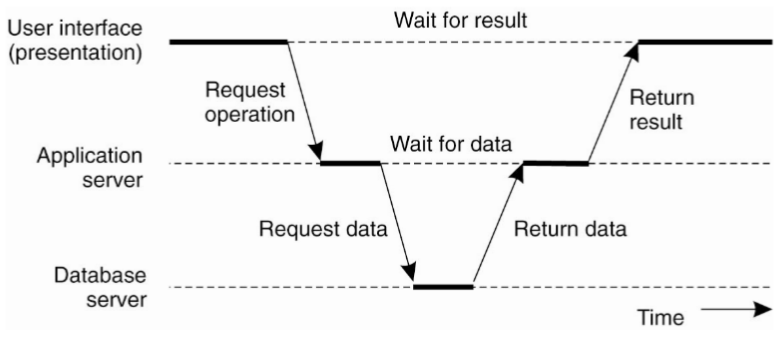

Figure 2.9: Three-tier Web Application

This is a good example in which the browser is on the client side, and the application and its database is on the server side:

$$\text{browser} \leftarrow | \rightarrow \text{http request} \longleftrightarrow \text{app server} \longleftrightarrow \text{database}$$

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. 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 serach 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 botton tier. The divide between user and server is not set in stone as we saw in the previous section.

2.4.4 Edge-Server Systems

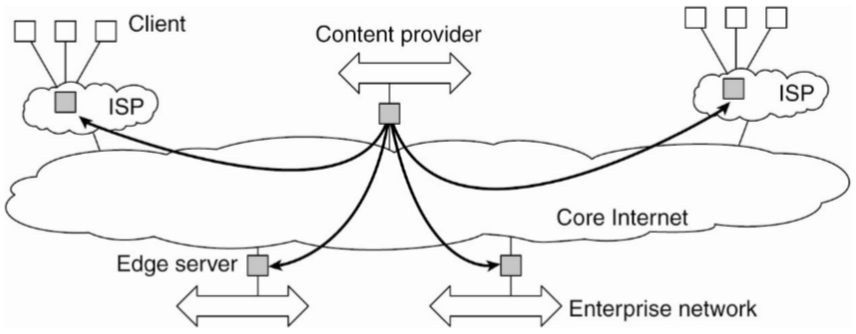

Figure 2.10: Edge Server

Now we look at a client-proxy-server. As the name suggests, there is an extra component in between. The proxy (labelled 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. An example is a proxy cache; this can lower latency under the assumption that it is closer to the client than the server. Many othe proxy services can be provided in addition to the caching. Content distribution networks (CDN) are a global network of proxy caches that that try to handle requests before heading to the server. An example is streaming applications.

2.5 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. They can also come as structured or unstructured systems. A peer can provide services and request services. Peers can also come and go at any time unlike a server that has to be there all the time.

An example of this architecture is a distributed hash table (DHT) service. This shows a structured P2P system as it looks like a ring (shown below). With a provided key, the system has to look up the value in the hash table. However, a very large hash table cannot be stored on one machine, so it is split into n pieces distribued across n machines. An example is chord, shown in the figure below.

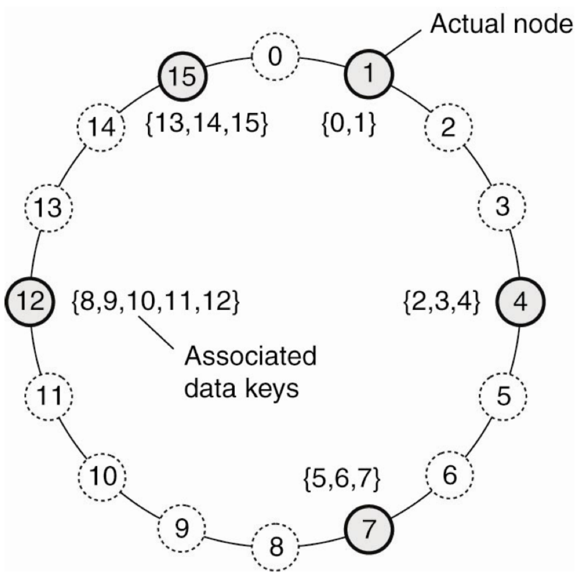

Figure 2.11: Chord Structure

This chord looks like a logical ring topology but the Internet is not a ring so it is an overlay topology. As mentioned before, the high level method used in these types of architectures is a key to value hash. Here, a search string is hashed into an int (key), and if there is a node in the system that matches the key then go to that node and download the file. In this example, there are $n = 16$ keys in the system. The darker circles are the nodes i.e. the peers.

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 [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 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.

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 "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.

More detail about chord can be found here:

https : //pdos.csail.mit.edu/papers/chord : sigcomm01/chord_sigcomm.pdf

2.5.1 Content Addressable Network (CAN)

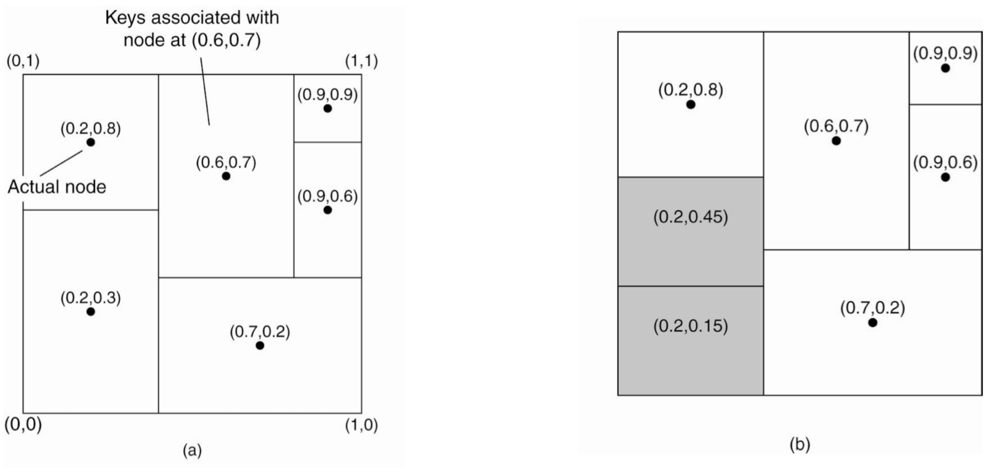

Figure 2.12: CAN Structure, with (b) showing a join procedure

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 examples, 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 ieach 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.

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.

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

2.5.2 Unstructured P2P Systems

Rather than adhering to some topological protocol such as a ring or a tree, unstructured topology is defined by randomized algorithms i.e. the network topology here 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 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.

The unstructured notion of such P2P systems framed early systems, but newer systems have more structue in order to reduce overhead.

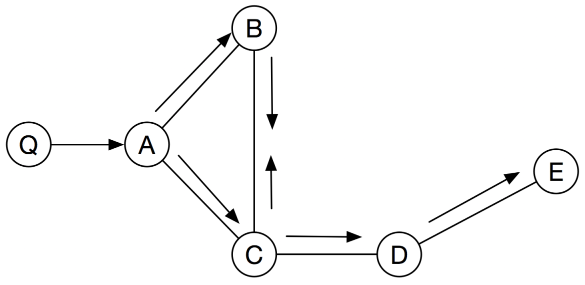

Figure 2.13: Search in Unstructured P2P System

From the figure above from last year's lecture, 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.

2.5.3 Structured and Unstructured P2P

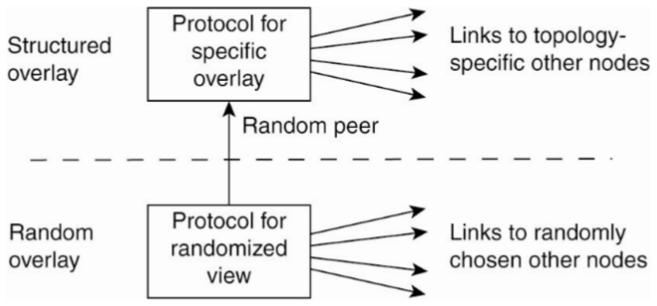
Structured overlay
Protocol for specific overlay
Links to topology-specific other nodes
Random peer
Random overlay
Protocol for randomized view
Links to randomly chosen other nodes

Figure 2.14: Structured and Unstructured P2P

In the above figure, we see that one can move from an unstructured system to a structured one.

Note: This slide was skipped in the lecture.

2.5.4 SuperPeers

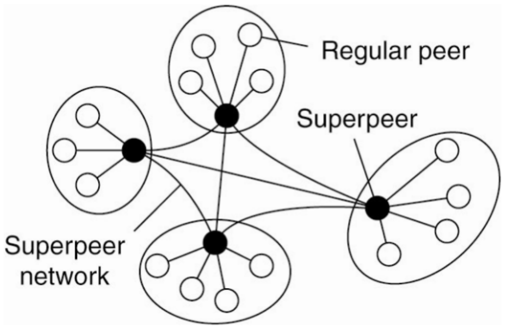

Figure 2.15: Graph with SuperPeer Structure

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.

In the past, Skype was a good example of how superpeers work. It tracked where users were and if they were logged in from a specific cluster. That was their P2P system, but now they have 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.

2.5.5 Collaborative Distributed Systems

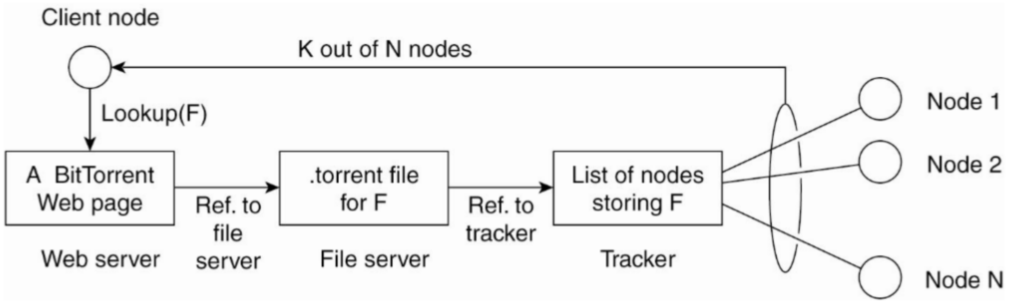

Figure 2.16: Collaborative Distributed Systems

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 that it connects to.

A system like BitTorrent can also take into account an altruism ratio, and slow down a node based on the ratio i.e. 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.

2 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 that 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 paramter in the system.

2.5.6 Other Topics

Note: The last two topics in the Lecture 2 slides were not covered in time. These topics are Self-Managing Systems and Feedback Control Model. They may be covered at the beginning of Lecture 3.