CHAPTER TWO
SYSTEM MODELS

2.1. Introduction

There are different ways on how to view the organization of a distributed system, but an obvious one is to make a distinction between the logical organization of the collection of software components and on the other hand the actual physical realization.

The organization of distributed systems is mostly about the software components that constitute the system. These software architectures tell us how the various software components are to be organized and how they should interact.

The actual realization of a distributed system requires that we instantiate and place software components on real machines. There are many different choices that can be made in doing so. The final instantiation of software architecture is also referred to as system architecture. In this chapter we will look into traditional centralized architectures in which a single server implements most of the software components (and thus functionality), while remote clients can access that server using simple communication means. In addition, we consider decentralized architectures in which machines more or less play equal roles, as well as hybrid organizations.

2.2. Architectural Styles

It refers to the logical organization of distributed systems into software components, also referred to as software architecture. Architectural style is formulated in terms of components, the way that components are connected to each other, the data exchanged between components and finally how these elements are jointly configured into a system. A component is a modular unit with well-defined required and provided interfaces that is replaceable within its environment; can be replaced provided that we respect its interfaces.

A somewhat more difficult concept to grasp is that of a connector, which is generally described as a mechanism that mediates communication, coordination, or cooperation among components. For example, a connector can be formed by the facilities for (remote) procedure calls, message passing, or streaming data.

Using components and connectors, we can come to various configurations, which, in tum have been classified into architectural styles. Several styles have by now been identified, of which the most important ones for distributed systems are:

1. Layered architecture
2. Object-based architecture
3. Data-centered architecture
4. Event-based architecture

❖ Layered architecture

The basic idea for the layered style is that components are organized in a layered fashion where a component at layer L_i is allowed to call components at the underlying layer L_{i-1}, but not the other way around. This model has been widely adopted by the networking community (i.e. network layer).

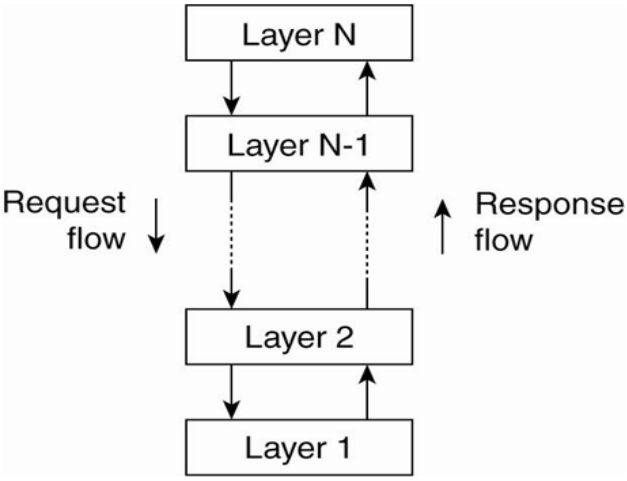

Figure 2.1: The layered architectural style

A key observation in this architecture is that control generally flows from layer to layer: requests go down the hierarchy whereas the results flow upward.

❖ Object-based architecture

In the object-based architecture, each object corresponds to a component and these components are connected through a (remote) procedure call mechanism. It matches the client-server system architecture.

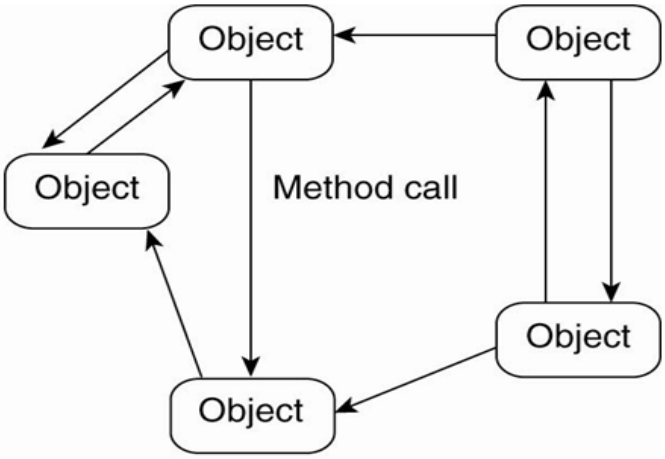

Figure 2.2: The object-based architectural style

❖ Data-centered architecture

Data-centered architectures evolve around the idea that processes communicate through a common (passive or active) repository like shared distributed file system. For example, web-based distributed systems are largely data-centric: processes communicate through the use of shared Web-based data services.

❖ Event-based architecture

In event-based architectures, processes essentially communicate through the propagation of events. For distributed systems, event propagation has generally been associated with what are known as publish/subscribe systems. The basic idea is that processes publish events after which the middleware ensures that only those processes that subscribed to those events will receive them. The main advantage of event-based systems is that processes are loosely coupled. In principle, they need not explicitly refer to each other. This is also referred to as being decoupled in space, or referentially decoupled.

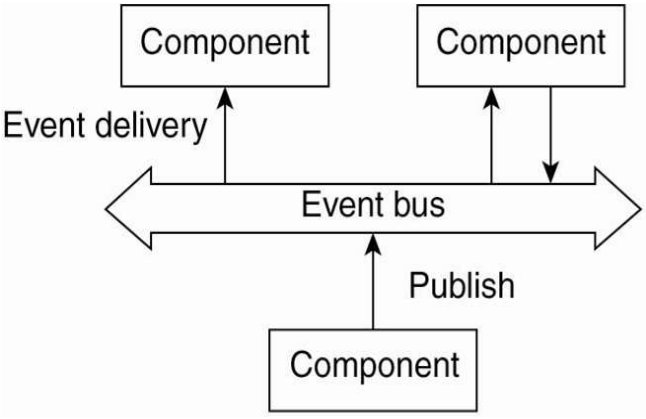

Figure 2.3: The event-based architectural style

Event-based architectures can be combined with data-centered architectures, yielding what is also known as shared data spaces. The essence of shared data spaces is that processes are now also decoupled in time: they need not both be active when communication takes place.

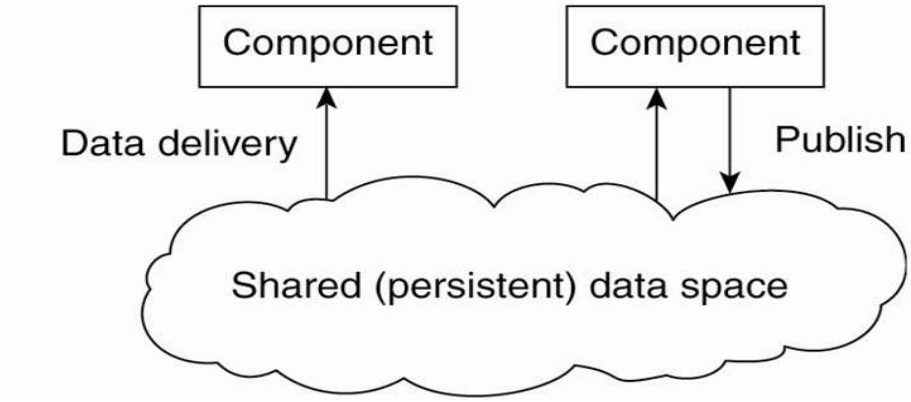

Figure 2.4: The shared data-space architectural style

2.3. System Architecture

System architecture refers to the logical organization of distributed systems into software components or how are processes organized in a system; where do we place software components. Deciding on software components, their interaction, and their placement is what system architecture is all about.

We will discuss centralized, decentralized and hybrid architectures.

2.3.1. Centralized Architectures

It is the idea thinking in terms of clients that request services from servers (client-server models). In the basic client-server model, processes in a distributed system are divided into two (possibly overlapping) groups. A server is a process implementing a specific service, for example, a file system service or a database service. A client is a process that requests a service from a server by sending it a request and subsequently waiting for the server's reply. This client-server interaction, also known as request-reply behavior is shown in the following figure.

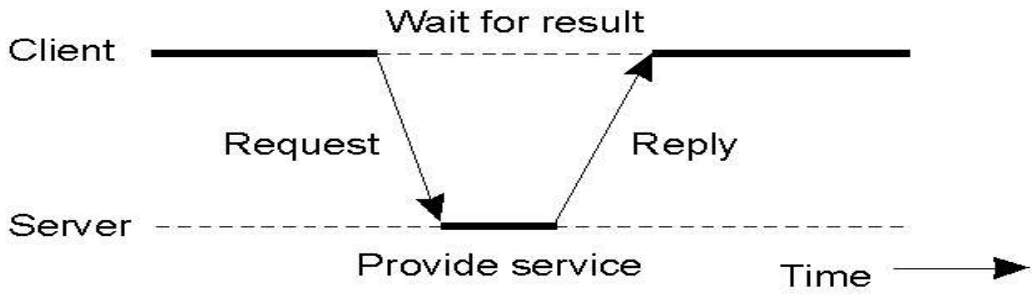

Figure 2.5: General interaction between a client and a server.

Communication between a client and a server can be implemented by means of a simple connectionless protocol when the underlying network is fairly reliable. In these cases, when a client requests a service, it simply packages a message for the server, identifying the service it wants, along with the necessary input data. The message is then sent to the server. The latter, in turn, will always wait for an incoming request, subsequently process it, and package the results in a reply message that is then sent to the client. Using a connectionless protocol has the obvious advantage of being efficient. As long as messages do not get lost or corrupted, the request/reply protocol just sketched works fine. Unfortunately, making the protocol resistant to occasional transmission failures is not trivial. The only thing we can do is possibly let the client resend the request when no reply message comes in. The problem, however, is that the client cannot detect whether the original request message was lost, or that transmission of the reply failed. If the reply was lost, then resending a request may result in performing the operation twice. If the operation was something like "transfer $10,000 from my bank account," then clearly, it would have been better that we simply reported an error instead. On the other hand, if the operation was "tell me how

much money I have left," it would be perfectly acceptable to resend the request. When an operation can be repeated multiple times without harm, it is said to be idempotent. Since some requests are idempotent and others are not it should be clear that there is no single solution for dealing with lost messages.

Many client-server systems use a reliable connection oriented protocol in which communication is inherently unreliable. For example, all Internet application protocols are based on reliable TCP/IP connections. In this case, whenever a client requests a service, it first sets up a connection to the server before sending the request. The server generally uses that same connection to send the reply message, after which the connection is torn down. The trouble is that setting up and terminating down a connection is relatively costly.

Application layering

The main issue in client-server model is that there is no clear distinction between a client and a server. For example, a server for a distributed database may act as a client when it forwards requests to different file servers responsible for implementing the database tables.

However, considering that many client-server applications are targeted toward supporting user access to databases, the following three levels exists.

i. The user-interface level:
ii. The processing level
iii. The data level

The user-interface level contains all that is necessary to directly interface with the user. Clients typically implement the user-interface level. This level consists of the programs that allow end users to interact with applications. The processing level contains applications. The data level manages the actual data that is being acted on.

For example, in an Internet search engine, the user interface is very simple: a user types in a string of keywords and is subsequently presented with a list of titles of Web pages. The back end is formed by a huge database of Web pages that have been pre fetched and indexed. The core of the search engine is a program that transforms the user's string of keywords into one or more database queries. It subsequently ranks the results into a list, and transforms that list into a series of HTML pages. Within the client-server model, this information retrieval part is typically placed at the processing level. The following figure shows this organization.

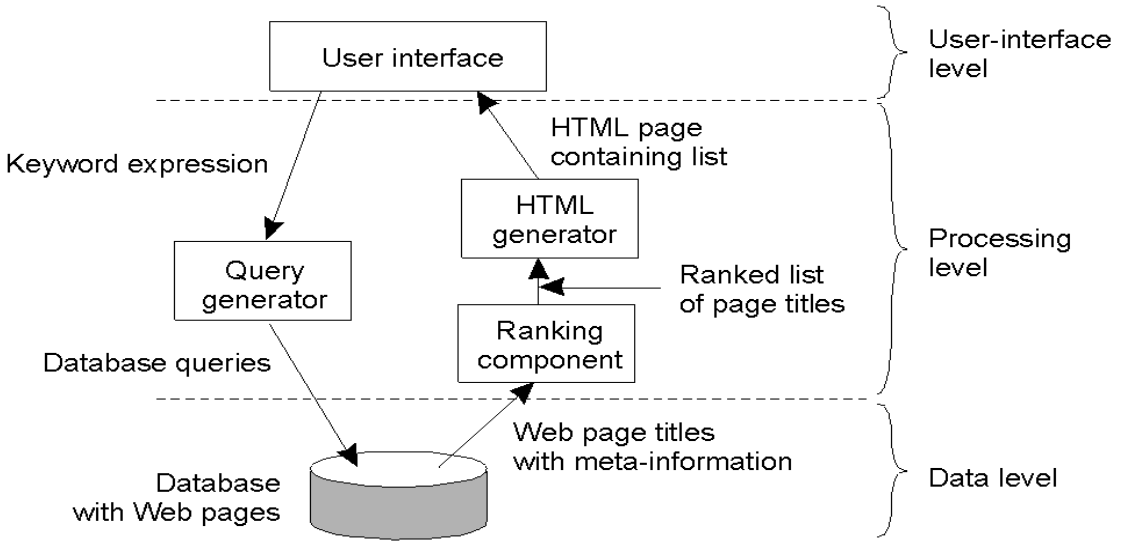

Figure 2.5: The organization of an Internet search engine into three different levels.

Multitiered Architectures

The distinction into three logical levels suggests possibilities for physically distributing a client-server application across several machines.

The simplest organization is to have only two types of machines:

i. A client machine containing only the programs implementing the user-interface level.
ii. A server machine containing the rest, that is the programs implementing the processing and data level.

In this organization everything is handled by the server while the client is essentially no more than a dumb terminal, possibly with a pretty graphical interface.

Another approach for organizing the clients and servers is to distribute the programs in the application layers across different machines, as shown in Fig. 2.6. As a first step, we make a distinction between only two kinds of machines: client machines and server machines, leading to what is also referred to as a (physically) two tiered architecture.

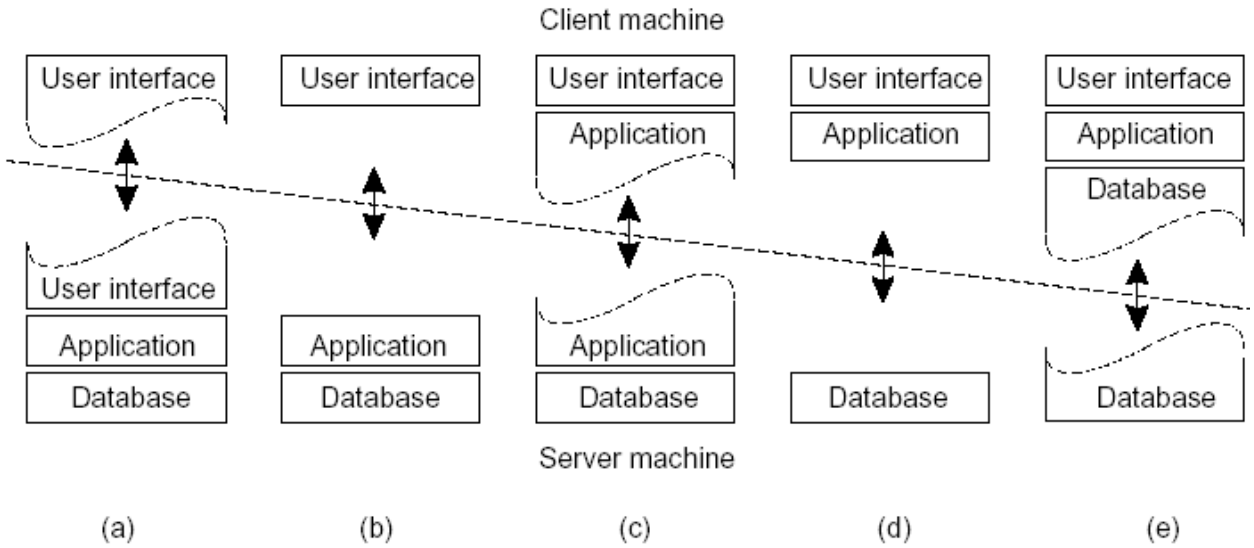

Figure 2.6: Two-tiered architecture: alternative client-server organizations

One possible organization is to have only the terminal-dependent part of the user interface on the client machine, as shown in Fig. 2.6 (a), and give the applications remote control over the presentation of their data. An alternative is to place the entire user-interface software on the client side, as shown in Fig. 2.6 (b). Continuing along this line of reasoning, we may also move part of the application to the front end, as shown in Fig. 2.6 (c). An example where this makes sense is where the application makes use of a form that needs to be filled in entirely before it can be processed. The front end can then check the correctness and consistency of the form, and where necessary interact with the user.

In many client-server environments, the organizations shown in Fig. 2.6 (d) and Fig. 2.6 (e) are particularly popular. These organizations are used where the client machine is connected through a network to a distributed file system or database. Essentially, most of the application is running on the client machine, but all operations on files or database entries go to the server. For example, many banking applications run on an end-user's machine where the user prepares transactions and such. Once finished, the application contacts the database on the bank's server and uploads the transactions for further processing. Fig. 2.6 (e) represents the situation where the client's local disk contains part of the data. For example, when browsing the Web, a client can gradually build a huge cache on local disk of most recent inspected Web pages.

Server-side solutions are becoming increasingly more distributed as a single server is being replaced by multiple servers running on different machines. In particular, when distinguishing only client and server machines, we miss the point that a server may sometimes need to act as a client, as shown in Fig. 2.7, leading to a (physically) three-tiered architecture.

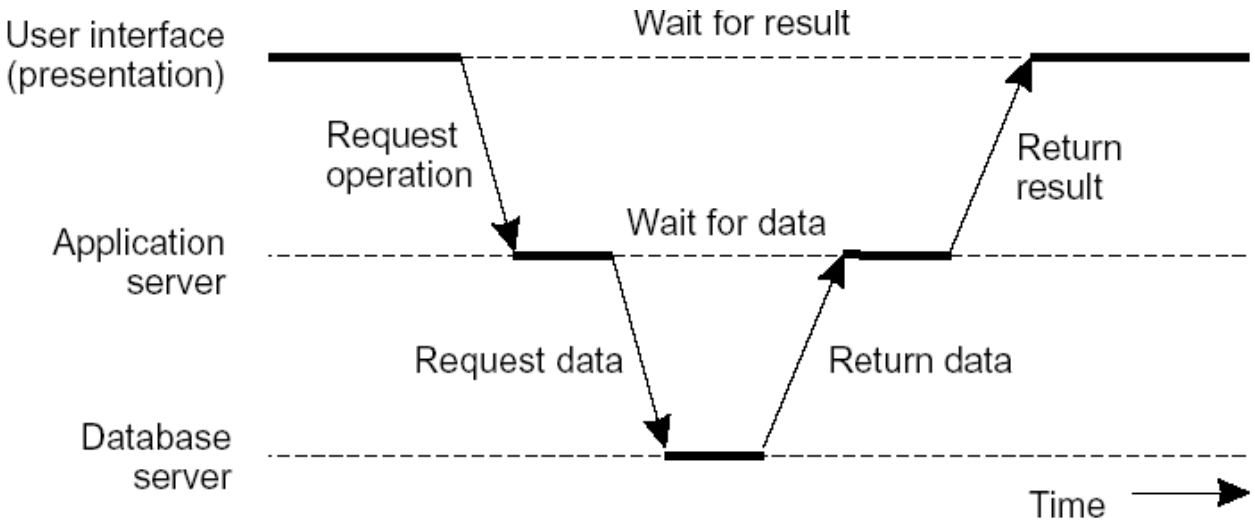

Figure 2.7: Three tiered architecture: an example of a server acting as a client

In this architecture, programs that form part of the processing level reside on a separate server, but may additionally be partly distributed across the client and server machines. An example where a three-tiered architecture is used is organization of Web sites. In this case, a Web server acts as an entry point to a site, passing requests to an application server where the actual processing takes place. This application server, in tum, interacts with a database server. For example, an application server may be responsible for running the code to inspect the available inventory of some goods as offered by an electronic bookstore. To do so, it may need to interact with a database containing the raw inventory data.

2.3.2. Decentralized Architectures

Multitiered client-server architectures are a direct consequence of dividing applications into a user-interface, processing components, and a data level. The different tiers correspond directly with the logical organization of applications. In many business environments, distributed processing is equivalent to organizing a client-server application as a multitiered architecture. We refer to this type of distribution as vertical distribution. The characteristic feature of vertical distribution is that it is achieved by placing logically different components on different machines. The term is related to the concept of vertical fragmentation as used in distributed relational databases, where it means that tables are split column-wise, and subsequently distributed across multiple machines.

Again, from a system management perspective, having a vertical distribution can help: functions are logically and physically split across multiple machines, where each machine is tailored to a specific group of functions. However, vertical distribution is only one way of organizing client-server applications.

In modern architectures, it is often the distribution of the clients and the servers that counts, which we refer to as horizontal distribution. In this type of distribution, a client or server may be physically split up into logically equivalent parts, but each part is operating on its own share of the complete data set, thus balancing the load. An example of horizontal distribution is a peer-to-peer system. The processes that constitute a peer-to-peer system are all equal. This means that the functions that need to be carried out are represented by every process that constitutes the distributed system. As a consequence, much of the interaction between processes is symmetric: each process will act as a client and a server at the same time (which is also referred to as acting as a servant).

Peer-to-peer architectures evolve around the question how to organize the processes in an overlay network, that is, a network in which the nodes are formed by the processes and the links represent the possible communication channels (which are usually realized as TCP connections). In general, a process cannot communicate directly with an arbitrary other process, but is required to send messages through the available communication channels. Two types of overlay networks exist: those that are structured and those that are not.

In a structured peer-to-peer architecture, the overlay network is constructed using a deterministic procedure. By far the most-used procedure is to organize the processes through a distributed hash table (DHT). In a DHT -based system, data items are assigned a random key from a large identifier space. Likewise, nodes in the system are also assigned a random number from the same identifier space. The crux of every DHT-based system is then to implement an efficient and deterministic scheme that uniquely maps the key of a data item to the identifier of a node based on some distance metric. Most importantly, when looking up a data item, the network address of the node responsible for that data item is returned. Effectively, this is accomplished by routing a request for a data item to the responsible node.

Unstructured peer-to-peer systems largely rely on randomized algorithms for constructing an overlay network. The main idea is that each node maintains a list of neighbors, but that this list is constructed in a more or less random way. Likewise, data items are assumed to be randomly placed on nodes. As a consequence, when a node needs to locate a specific data item, the only thing it can effectively do is flood the network with a search query.

2.3.3. Hybrid Architectures

These classes of distributed system architectures are the one in which client-server solutions are combined with decentralized architectures.

An important class of distributed systems that is organized according to a hybrid architecture is formed by edge-server systems. These systems are deployed on the Internet where servers are placed "at the edge" of the network. This edge is formed by the boundary between enterprise networks and the actual Internet, for example, as provided by an Internet Service Provider (ISP). Likewise, where end users at home connect to the Internet through their ISP, the ISP can be considered as residing at the edge of the Internet. This leads to a general organization as shown in the following figure, Figure 2.8.

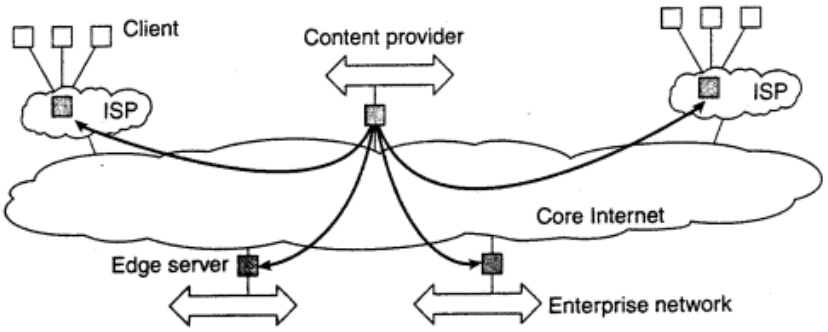

Figure 2.8: Viewing the Internet as consisting of a collection of edge servers.

End users, or clients in general, connect to the Internet by means of an edge server. The edge server's main purpose is to serve content, possibly after applying filtering and transcoding functions. More interesting is the fact that a collection of edge servers can be used to optimize content and application distribution. The basic model is that for a specific organization, one edge server acts as an origin server from which all content originates. That server can use other edge servers for replicating Web pages.

Another instance where there is hybrid structures is that of collaborative distributed systems. The main issue in many of these systems to first get started, for which often a traditional client-server scheme is deployed. Once a node has joined the system, it can use a fully decentralized scheme for collaboration. We can consider the BitTorrent file-sharing system in this case. BitTorrent is a peer-to-peer file downloading system. Its principal working is shown in Figure 2.9. The basic idea is that when an end user is looking for a file, he downloads chunks of the file from other users until the downloaded chunks can be assembled together yielding the complete file. An important design goal was to ensure collaboration. In most file-sharing systems, a significant fraction of participants merely download files but otherwise contribute close to nothing. To this end, a file can be downloaded only when the downloading client is providing content to someone else.

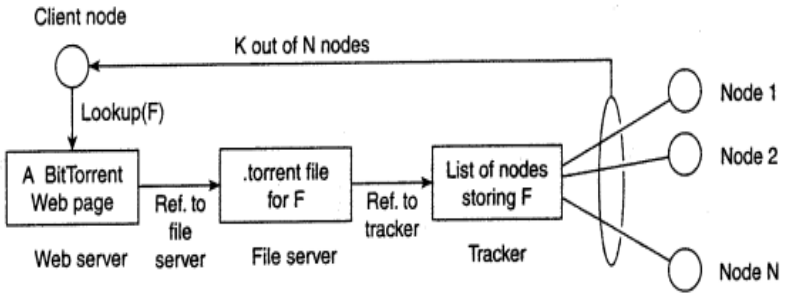

Figure 2.9. The principal working of BitTorrent

To download a file, a user needs to access a global directory, which is just one of a few well-known Web sites. Such a directory contains references to what are called .torrent files. A .torrent file contains the information that is needed to download a specific file. In particular, it refers to what is known as a tracker, which is a server that is keeping an accurate account of active nodes that have (chunks) of the requested file. An active node is one that is currently downloading another file. Obviously, there will be many different trackers, although (there will generally be only a single tracker per file (or collection of files).

Once the nodes have been identified from where chunks can be downloaded, the downloading node effectively becomes active. At that point, it will be forced to help others, for example by providing chunks of the file it is downloading that others do not yet have. This enforcement comes from a very simple rule: if node P notices that node Q is downloading more than it is uploading, P can decide to decrease the rate at which it sends data to Q. This scheme works well provided P has something to download from Q. For this reason, nodes are often supplied with references to many other nodes putting them in a better position to trade data. Clearly, BitTorrent combines centralized with decentralized solutions. As it turns out, the bottleneck of the system is, not surprisingly, formed by the trackers.

Further reading assignment

1. Pure Layered, Mixed Layered and Layered with up calls organization
2. RESTful Architecture and (Service Oriented Architecture-SOA)
3. Applications that work based on Client-Server, Peer-to-Peer and Hybrid architecture
4. Middleware Organization (Wrappers, Interceptors and Modifiable middleware)