Lecture 34: Distributed Computing

Definitions

- *Sequential Process:* Our subject matter up to now: processes that (ultimately) proceed in *a single sequence* of primitive steps.
- *Concurrent Processing:* The logical or physical division of a process into multiple sequential processes.
- *Parallel Processing:* A variety of concurrent processing characterized by the *simultaneous execution* of sequential processes.
- *Distributed Processing:* A variety of concurrent processing in which the individual processes are physically separated (often using heterogeneous platforms) and communicate through some network structure.

Purposes

We may divide a single program into multiple programs for various reasons:

- *Computation Speed* through operating on separate parts of a problem simultaneously, or through
- *Communication Speed* through putting parts of a computation near the various data they use.
- *Reliability* through having mulitple physical copies of processing or data.
- *Security* through separating sensitive data from untrustworthy users or processors of data.
- *Better Program Structure* through decomposition of a program into logically separate processes.
- *Resource Sharing* through separation of a component that can serve mulitple users.
- *Manageability* through separation (and sharing) of components that may need frequent updates or complex configuration.

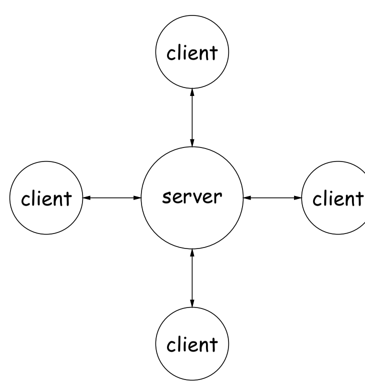
Communicating Sequential Processes

- All forms of concurrent computation can be considered instances of *communicating sequential processes*.
- That is, a bunch of "ordinary" programs that communicate with each other through what is, from their point of view, input and output operations.
- Sometimes the actual communication medium is *shared memory:* input looks like reading a variable and output looks like writing a variable. In both cases, the variable is in memory accessed by multiple computers.
- At other times, communication can involve I/O over a network such as the Internet.
- In principle, either underlying mechanism can be made to look like either access to variables or explicit I/O operations to a programmer.

Distributed Communication

- With sequential programming, we don't think much about the cost of "communicating" with a variable; it happens at some fixed speed that is (we hope) related to the processing speed of our system.
- With distributed computing, the architecture of communication becomes important.
- In particular, costs can become uncertain or heterogeneous:
  - It may take longer for one pair of components to communicate than for another, or
  - The communication time may be unpredictable or load-dependent.

Simple Client-Server Models



- Example: web servers
- Good for providing a service
- Many clients, one server
- Easy server maintenance.
- Single point of failure
- Problems with scaling

Variations: on to the cloud

- Google and other providers modify this model with redundancy in many ways.
- For example, *DNS load balancing* (DNS = *Domain Name System*) allows us to specify multiple servers.
- Requests from clients go to different servers that all have copies of relevant information.
- Put enough servers in one place, you have a *server farm*. Put servers in lots of places, and we have a *cloud*.

Communication Protocols

- At the lowest level, computers, pads, laptops, and phones (the data facilities, that is) are able to send out and receive arbitrary streams of bits into some network.
- Not very useful unless there is some agreement (protocol) as to what these bits are supposed to mean.

Protocol Layers

- In fact, we use a whole *stack* of protocol layers, each using the layer below, and each providing some *communication abstraction*:
  - The *IP Protocol* is provides a way to specify destinations and send *raw segments of messages* to those destinations, with no guarantee of delivery.
  - Inconvenient to deal with this low level directly, so the *TCP* (Transmission Control Protocol), built on top of IP, provides the abstraction of sending a complete message reliably. The software takes care of breaking the message into segments, sending those segments (via IP), reassembling them in order on the other end, and seeing that they have been correctly received.
  - The DNS (Domain Name Service), built on IP and TCP, is a distributed database that handles conversions between human readable names (URLs), such as (http://cs61a.org) and IP addresses (e.g. 104.199.121.146).
  - The *HyperText Transfer Protocol* (HTTP), built on TCP, handles transfer of requests and responses from web servers.

Example: HTTP

- When you click on a link, such as the syllabus:

  `http://cs61a.org/articles/about.html`

  your browser:
  - Consults the DNS to find out the IP address for `cs61a.org`.
  - Sends a message to port 80 at that address:

    ```
    GET articles/about.html HTTP 1.1
    ```

  - The program listening there (the web server) then responds with

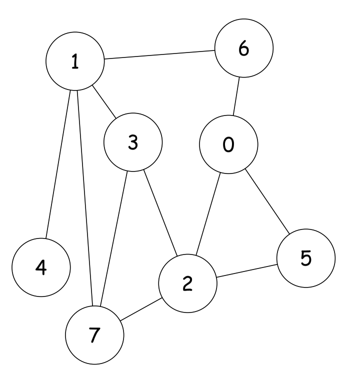
    ```
    HTTP/1.1 200 OK
    Content-Type: text/html
    Content-Length: 1354

    <html> ... text of web page
    ```

- Protocol has other messages: forexample, POST is often used to send data in forms from your browser. The data follows the POST message and other headers.

Still Another Level Of Abstraction: Sessions

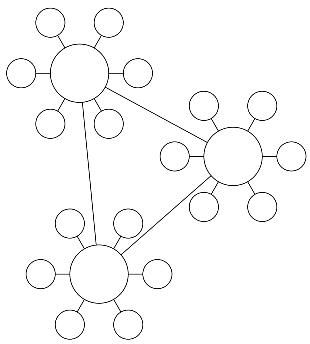
- The HTTP protocol was originally conceived as *stateless*: the server to which your browser sends messages does not reliably know who sends the messages and therefore which messages are part of the same conversation.
- So, early on, developers created an *abstraction* of a conversation ("session") on top of HTTP.
- A message from the server can contain *cookies:* pieces of data that the browser retains, and sends back to the server whenever it sends a message to the same address.
- For example, a cookie can hold a *session id*, a number, created randomly, that the browser sends back to the server so that the server can use it as a key to retrieve the state of the conversation.
- Alternatively, a server can send the actual state of the conversation to the browser and let the browser store it and send it back. (Have to be careful, or this can be a pathway to subverting the server).

Peer-to-Peer Communication



- No central point of failure; clients talk to each other.
- Can route around network failures.
- Computation and memory shared.
- Can grow or shrink as needed.
- Used for file-sharing applications, botnets (!).
- But, deciding routes, avoiding congestion, can be tricky.
- (E.g., Simple scheme, broadcasting all communications to everyone, requires N^2 communication resource. Not practical.
- Maintaining consistency of copies requires work.
- Security issues.

Clustering



- A peer-to-peer network of "supernodes," each serving as a server for a bunch of clients.
- Allows scaling; could be nested to more levels.
- Examples: Skype, network time service.