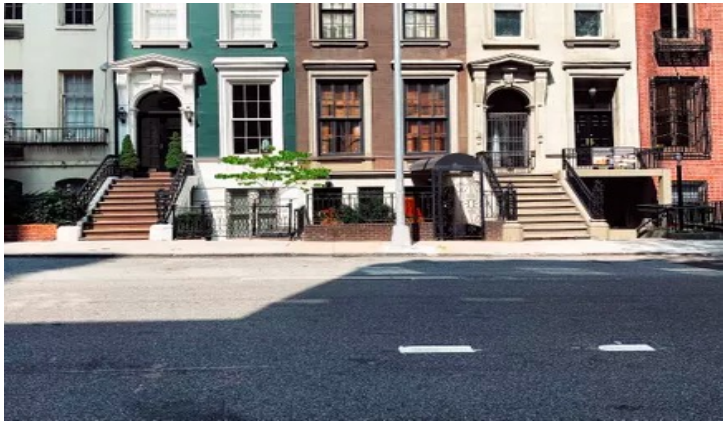
SE 211
Software Specification and Design II

Styles in Software Architecture

Architectural Styles



Architectural Styles

- An architectural style describes a method of construction and a set of features that make it notable.

Architectural Styles

- An architectural style describes a method of construction and a set of features that make it notable.

- A software architectural style defines a family of systems in term of structural organization.

  Components and connectors are combined in specific ways and reflect design decisions given the objectives and constraints of the problem to be solved.

Basic Properties of Styles

- The design elements
  - Component and connector types; data elements (e.g., objects, servers, pipes, filters)
- The configuration rules
  - Topological constraints that determine allowed compositions of elements (e.g., a component may be connected to at most two other components)
- A semantic interpretation
  - Compositions of design elements have well-defined meanings

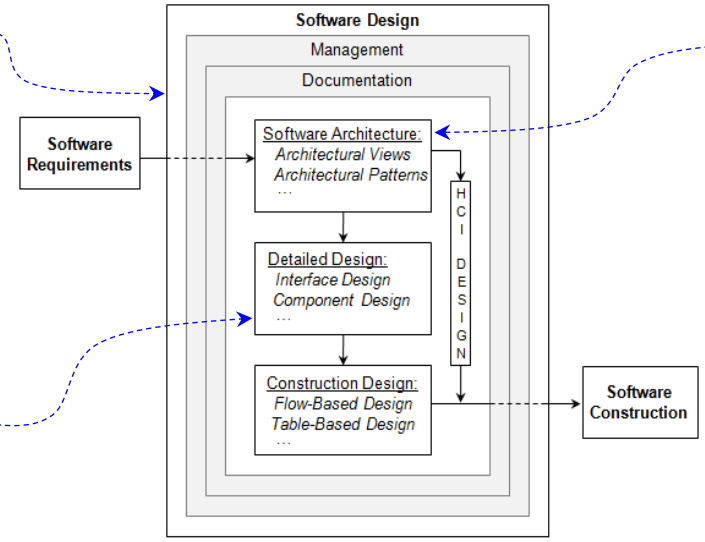
Architecture Styles vs. Patterns

- The terms *architectural styles* and *architectural patterns* are used interchangeably to denote reusable design solutions that occur during the *software architecture activity* of the design process.
- Architectural styles and architectural patterns do not describe the detailed design of systems
  - They are used as basis for system decomposition and for analyzing the structure of systems in principled manner.
- The term *Design Patterns* is used to denote reusable design solutions that occur during the *detailed design activity* of the design process.

Architecture Styles vs. Patterns

Software design model presented in Chapter 1

Architectural Styles and Patterns exist here!



Design Patterns exist here

Benefits of Using Styles

- Design reuse
  - Well-understood solutions applied to new problems
- Code reuse
  - Shared implementations of invariant aspects of a style
- Understandability of system organization
  - A phrase such as “client-server” conveys a lot of information

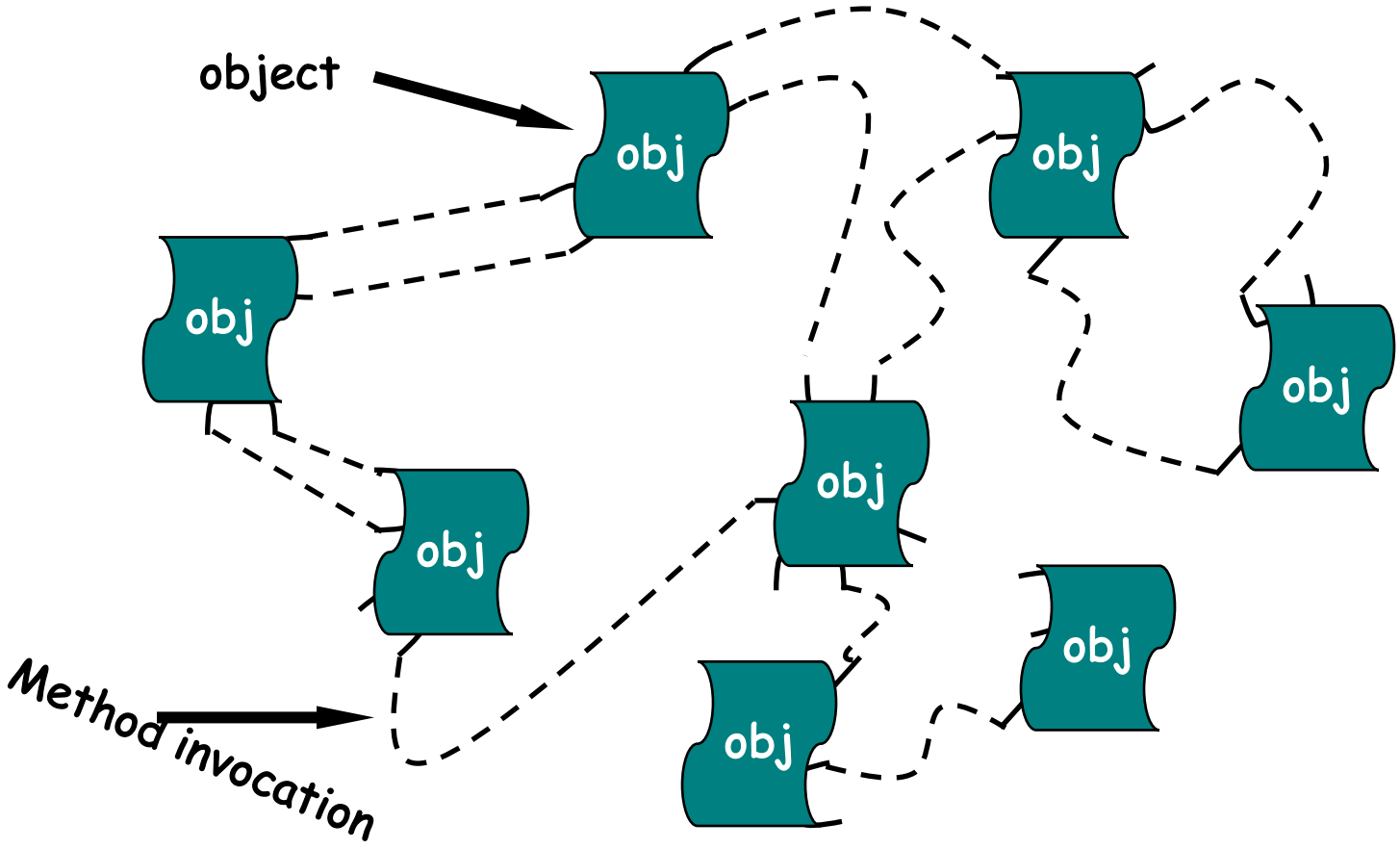
Some Common Styles

- Traditional, language-influenced styles
  - Object-oriented
  - Main program and subroutines
- Layered
  - Client-server
  - Web applications
  - Hierarchical
- Data-flow
  - Pipe and filter
- Shared memory
  - Blackboard
  - Rule based
- Interpreter
  - Interpreter
  - Mobile code
- Implicit invocation
  - Event-based
  - Publish-subscribe
- Peer-to-peer

Language-Influenced Styles

- Object Oriented
- Main Program and Subroutines

Object-Oriented Style



Object-Oriented Style (cont'd)

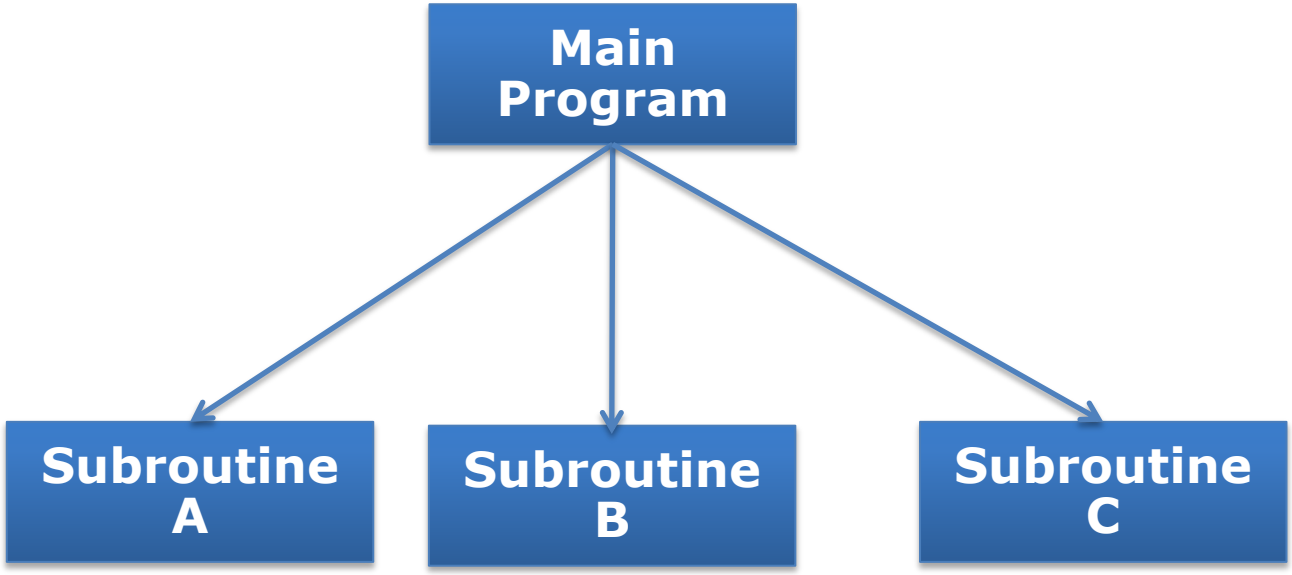
- Components: objects
  - Data and associated operations
- **Connectors**: method invocations
- Data: arguments passed to methods
- State strongly encapsulated. Internal representation is hidden from another object.
- Objects are responsible for their internal representation integrity.
- Topology can vary arbitrarily: data and interfaces can be shared through inheritance.

Object-Oriented Style (cont'd)

- Advantages
  - Integrity: data is manipulated only be appropriate methods.
  - Abstraction: internals are hidden.
  - Suitable for applications in which a central issue is identifying and protecting related bodies of information (data).

- Disadvantages
  - Not efficient enough for high performance computing (e.g., scientific computing, data science)
  - Distributed applications requires extensive middleware support

Main Program and Subroutines



Main Program and Subroutines (cont'd)

- Decomposition based upon separation of functional processing steps
- Components: main program and subroutines
- **Connectors**: function/procedure calls
- Data: values passed in/out subroutines
- Static organization is hierarchical

Main Program and Subroutines (cont'd)

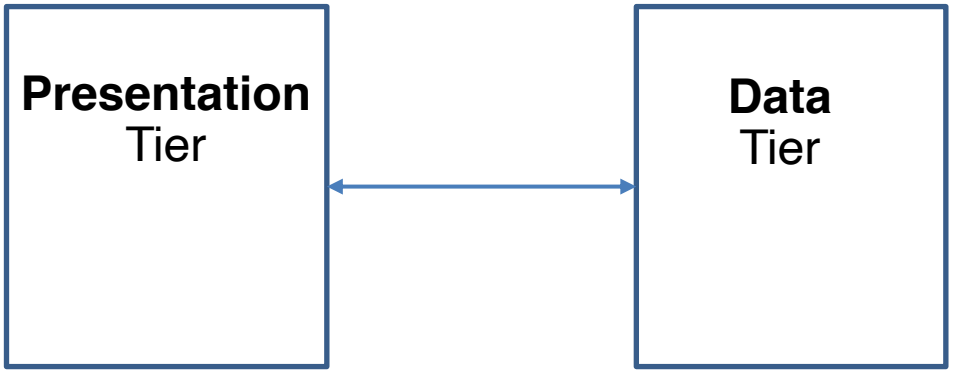
- Advantages
  - Large established base (has been used in the development of a large number of existing systems).
  - Applicable to all procedural programming languages.

- Disadvantages
  - Different than the object-oriented paradigm.

Client-Server Style

- The essence of the client-server style is that two kinds of processes (programs) that are specialized for different tasks, running on possibly different environments (hardware, operating systems, and applications software) co-operate to solve a computing problem.
- One process (the client) *makes a request*.
- The other process (the server) *performs* the service requested (if the request is a valid one).
- For example: news readers, e-mail, online banking, …

2-Tier Architecture



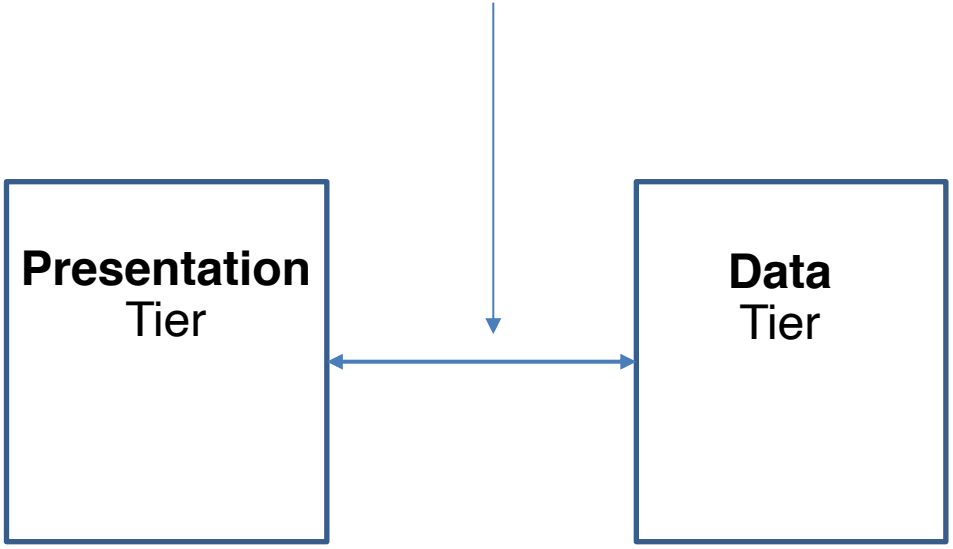
Where information is collected from and presented to.

Where information is stored and managed.

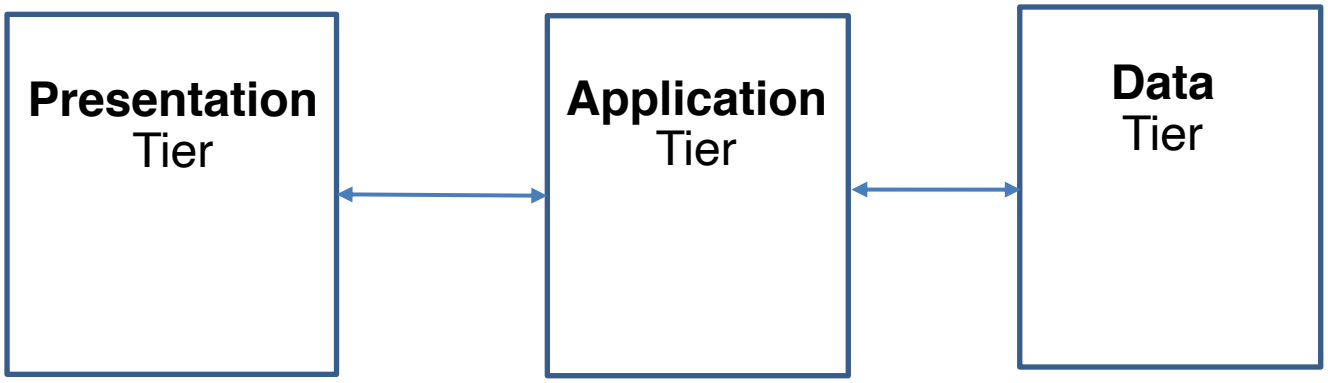
Business logic resides in either tier or both.

2-Tier Architecture

If this is over a network, we have a distributed system or distributed architecture.



3-Tier Architecture



Versions of Distributed Client-Server

Roll your own Distributed Program

Distributed Objects

Client-Server Over Web Protocols

Web Applications

- A Web application (web app) is a client-server program with the following characteristics:
- The code that implements the ***client*** runs in a browser (unlike programs that run natively on an operating system)
- The code that implements the ***server*** resides on a computer connected to the network, although it can reside on the local machine.
- The client and the server exchange information using the Hyper Text Transfer Protocol (HTTP).

Basics of a Web Application

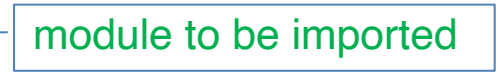
- The end-user enters in the browser a URL (typically, the address of a computer connected to the network, although it can be the local machine).
  - The computer identified by the URL is running a program called the *web server*. The web server is waiting to receive information.
- The browser uses the URL entered by the end-user to send a request for information to the web server.
- The web server receives the request and returns information to the browser.
- The browser interprets and executes the information received from the web server.
- From that point on, the end-user interacts with the program that is running in the browser, and the exchange of information between the *client* and *server* continues.

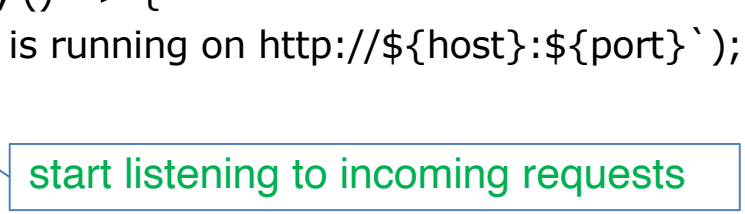
A Simple Server: hello-server.js

```
const http = require("http");
const host = 'localhost';
const port = 8000;


const requestListener = function (req, res) {
    res.writeHead(200);         // response header
    res.end("Hello World!");    // response content
};


const server = http.createServer(requestListener);
server.listen(port, host, () => {
    console.log(`Server is running on http://${host}:${port}`);
});
```





Starting the Server

Command to start the server:

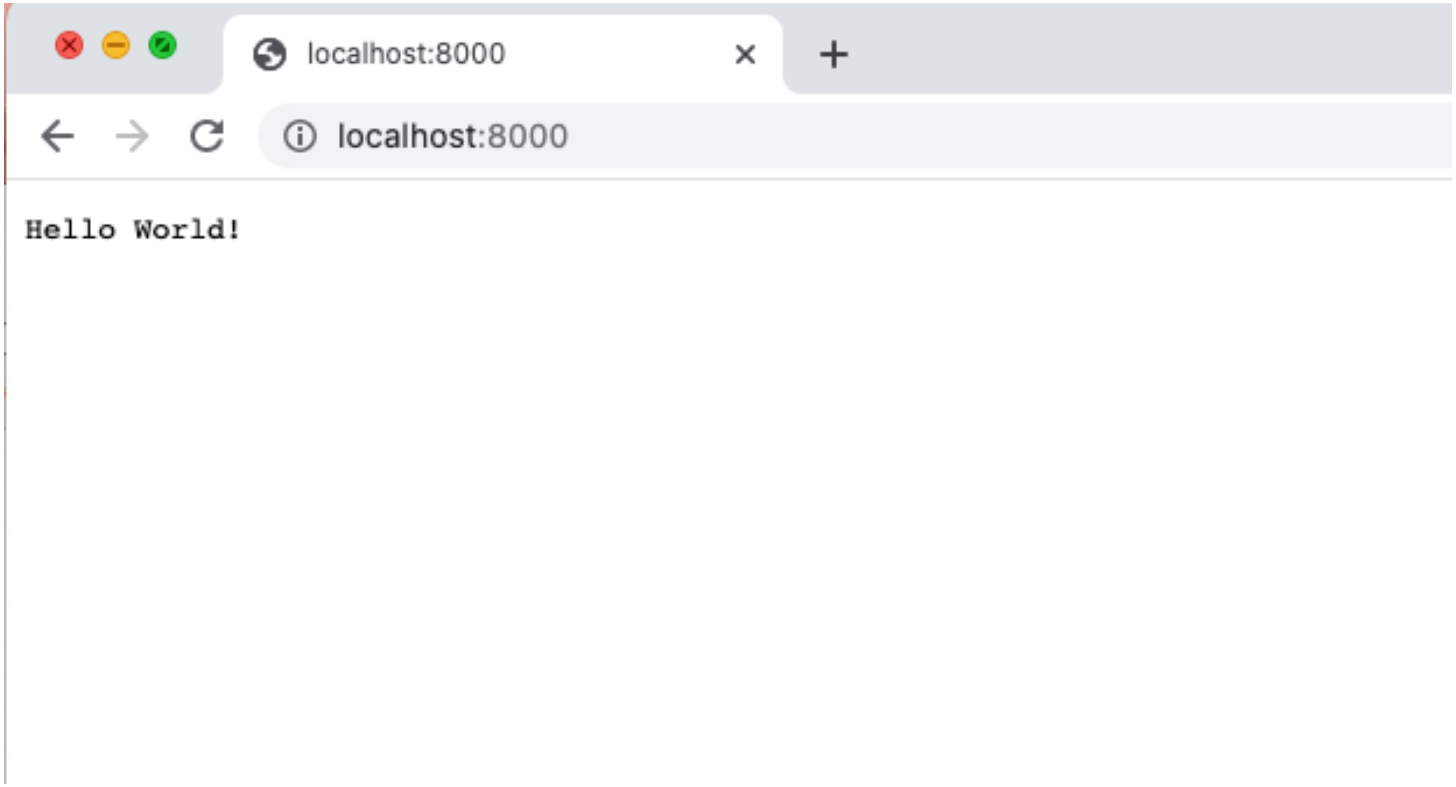
```
$ node hello-server.js
```

Note: This requires to download and install Node.js

Output displayed on stdout:

```
Server is running on http://localhost:8000
```

Contacting the Server



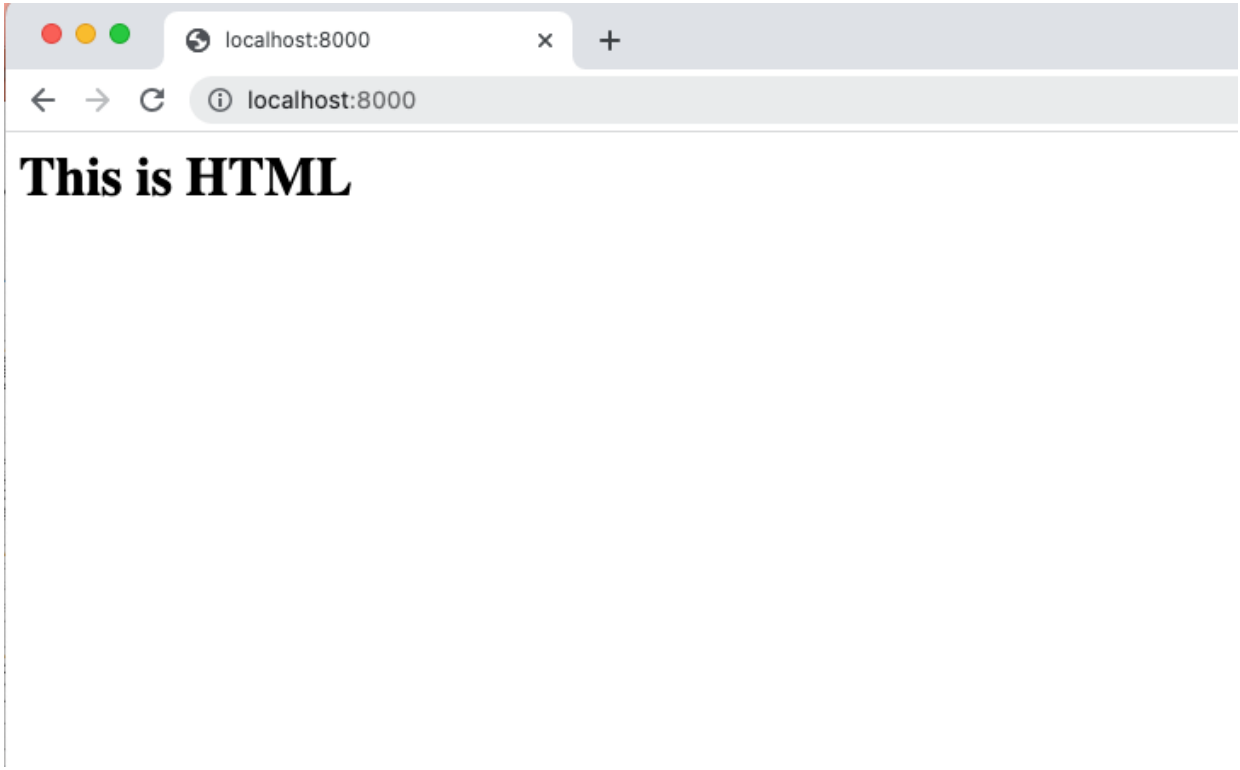
A Simple Server: html-server1.js

```
const http = require("http");

const host = 'localhost';
const port = 8000;

const requestListener = function (req, res) {
    res.setHeader("Content-Type", "text/html");
    res.writeHead(200);
    res.end(`<html><body><h1>This is HTML</h1></body></html>`);
};

const server = http.createServer(requestListener);
server.listen(port, host, () => {
    console.log(`Server is running on http://${host}:${port}`);
});
```


localhost:8000
localhost:8000
This is HTML

A Simple Server: html-server2.js

```
const http = require("http");

const host = 'localhost';
const port = 8000;

const fs = require('fs').promises;

const requestListener = function (req, res) {
    fs.readFile(__dirname + "/index.html")
        .then(contents => {
            res.setHeader("Content-Type", "text/html");
            res.writeHead(200);
            res.end(contents);
        })
        .catch(err => {
            res.writeHead(500);
            res.end(err);
            return;
        });
};
```

A Simple Server: html-server2.js (cont'd)

```
const server = http.createServer(requestListener);
server.listen(port, host, () => {
    console.log(`Server is running on http://${host}:${port}`);
});
```

index.html

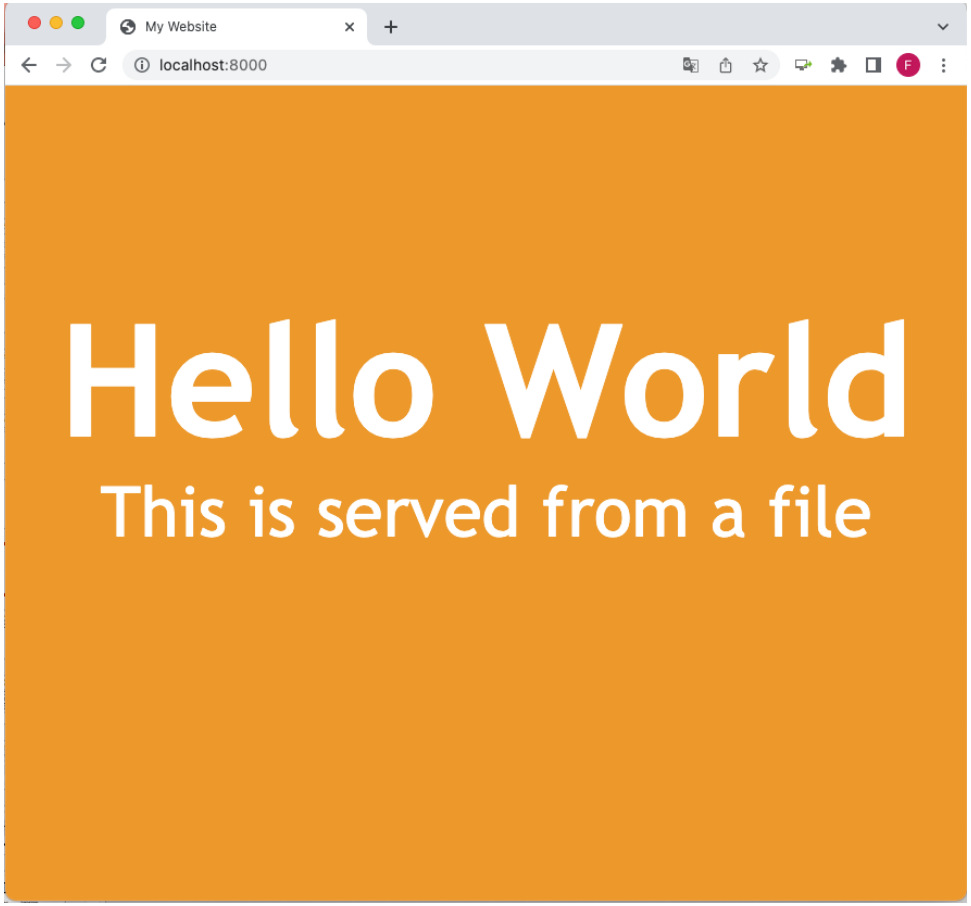
```
<!DOCTYPE html>

<head>
   <title>My Website</title>
   <style>
            // styling statements go here
            …
   </style>
</head>

<body>
   <div class="center">
      <h1>Hello World</h1>
      <p>This is served from a file</p>
   </div>
</body>

</html>
```

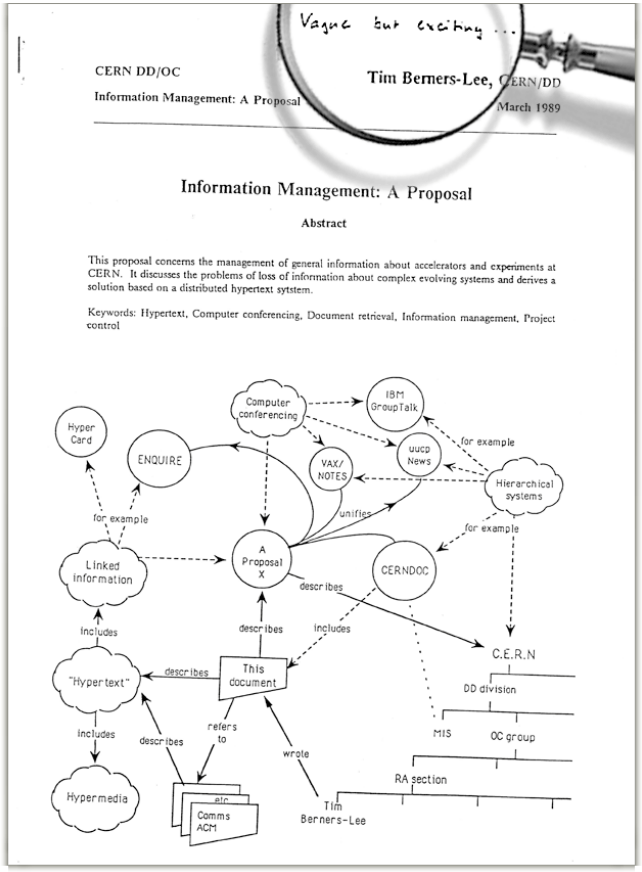
Displaying the Contents of index.html



Web Applications and the WWW

- The architecture of web applications is tightly coupled with the evolution of the World Wide Web.
- The evolution of web is one of the best examples of architecture done right.
- Two different generations of WWW: Web 1.0 and Web 2.0
- Perhaps soon Web 3.0.

History of the Web – circa 1989/1990

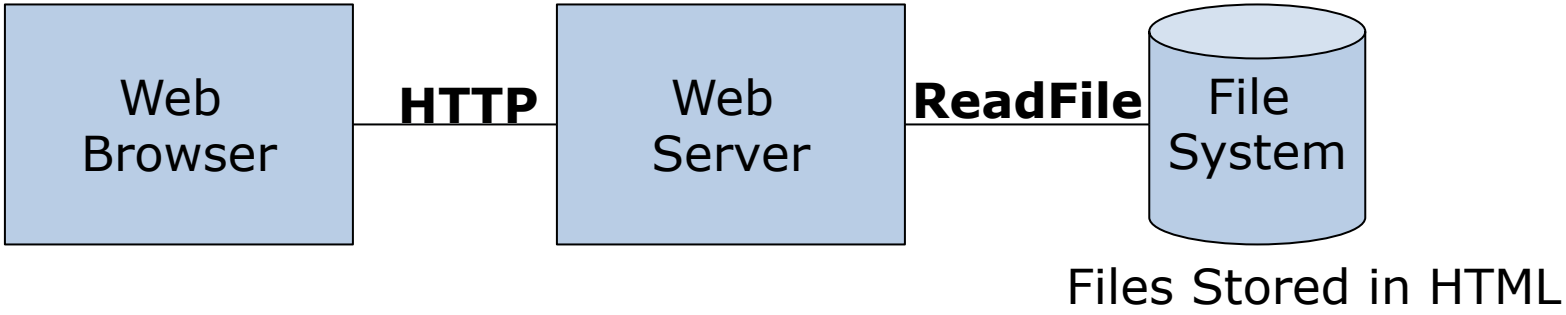


- In March 1989, Tim Berners-Lee wrote a proposal that outlined the architecture of the web
- Initial proposal was considered vague and not well received, it was revised and eventually accepted in October 1990
- Introduced the concepts we know today:
  - HTML: Formatting/Markup
  - URI: Addressing approach (now generally called URL)
  - HTTP: The protocol for encoding and transmitting information over the web

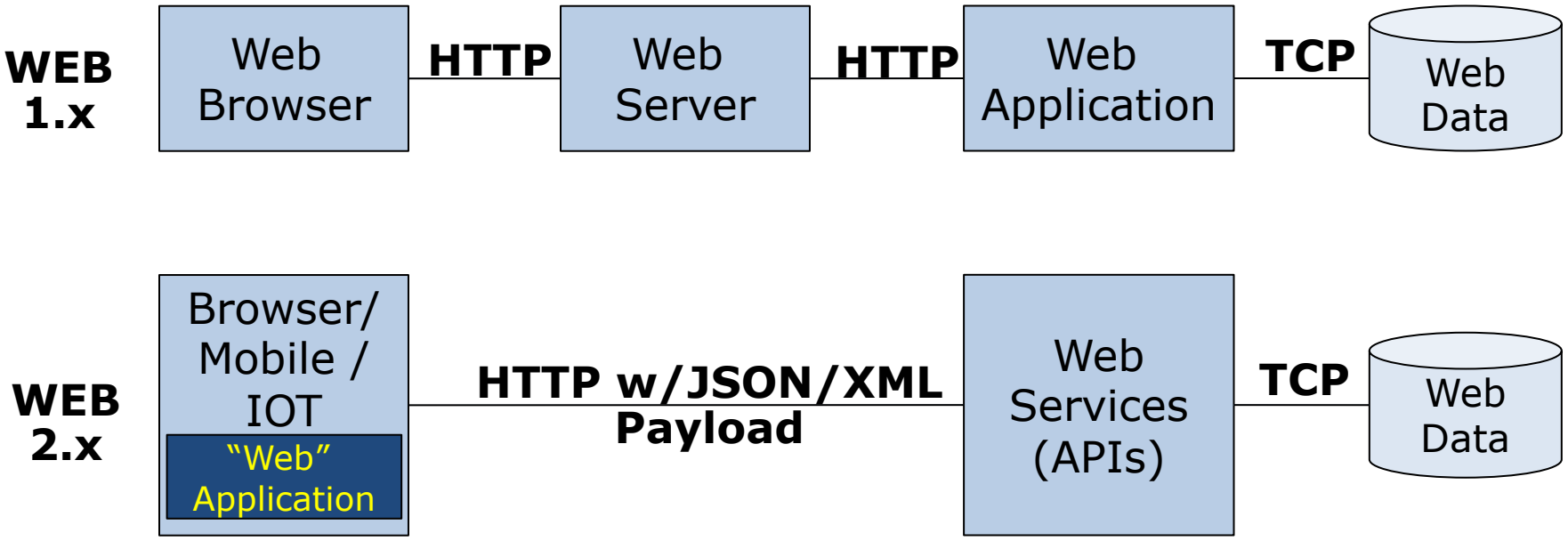
Web 1.0 Summary (from an architecture perspective)

- Web 1.0 existed for about 15 years [1990 – 2005]
- It started out supporting read-only content
- It evolved to supporting reasonable applications, that could run at reasonable scale over the basic web architecture
- The underlying architecture changed very little over this period, mainly supporting enhancements that allowed content-rich applications to be developed
- Introduced the MVC pattern as a best practice for developing web-centric applications

Web 1.0 – The "Read Only Web" Architecture

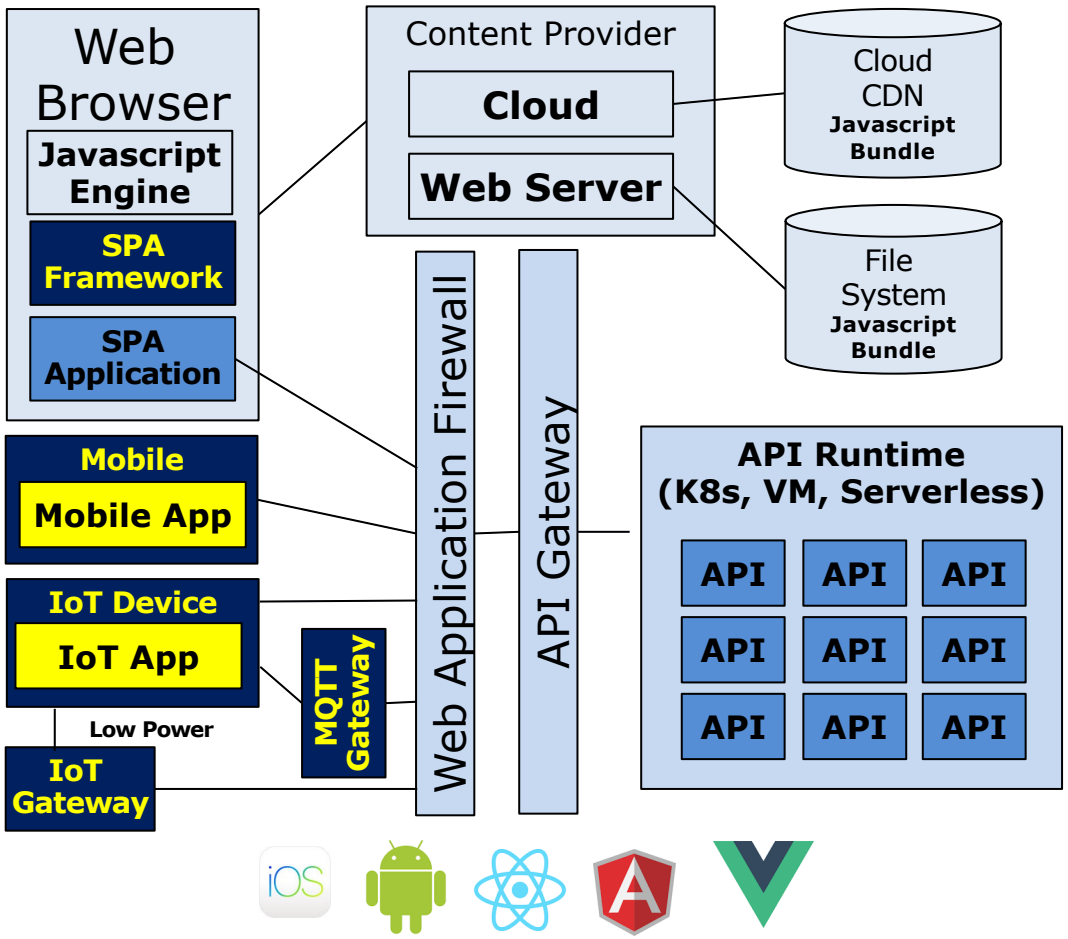


Web 1.x vs. Web 2.x



- Web 1.x was about the evolution from static content to dynamic content.
- Web 2.x is about running fully featured applications on the client. Client types expand from just a browser to mobile and IoT.

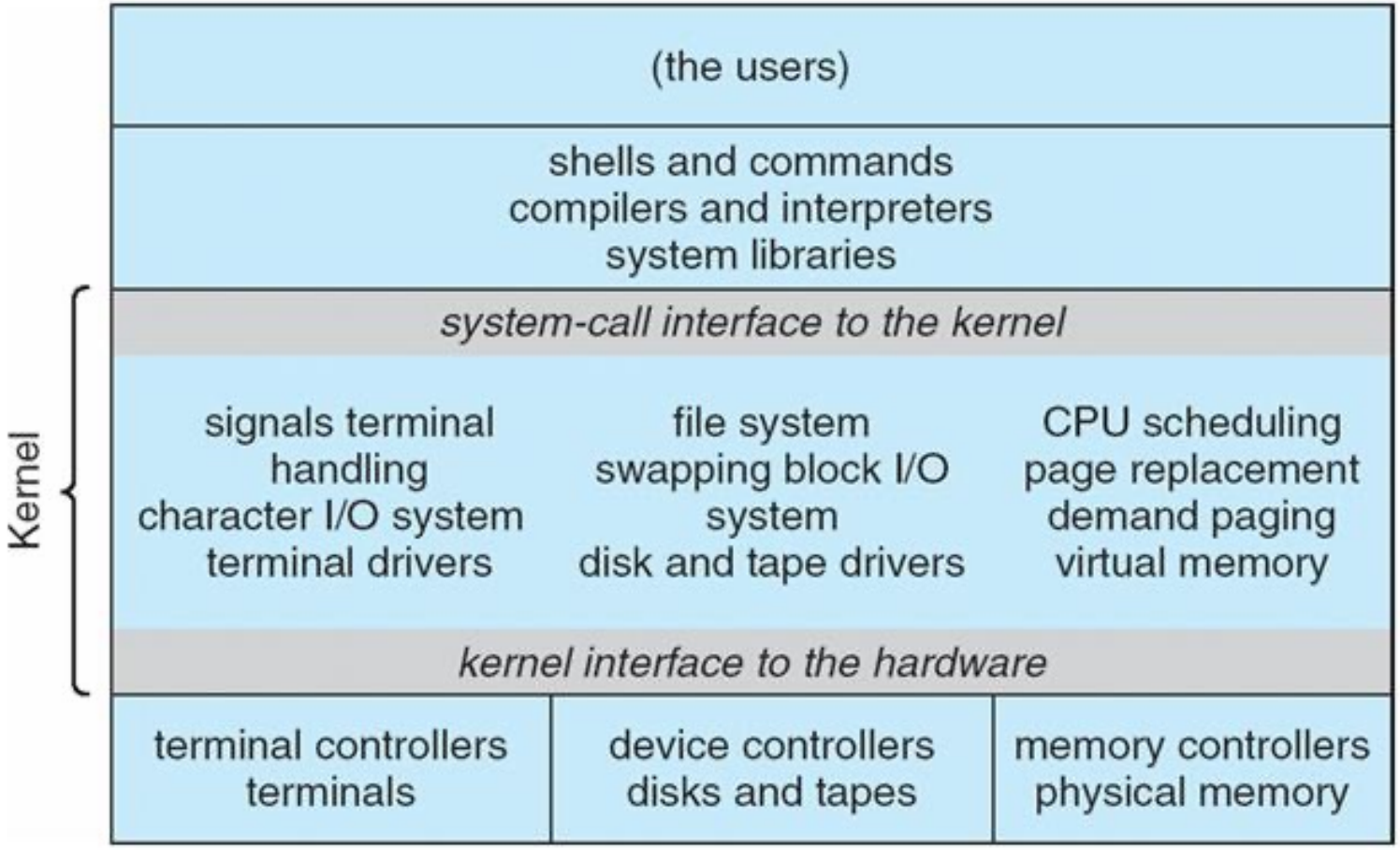
Web 2.x – Circa 2008-today



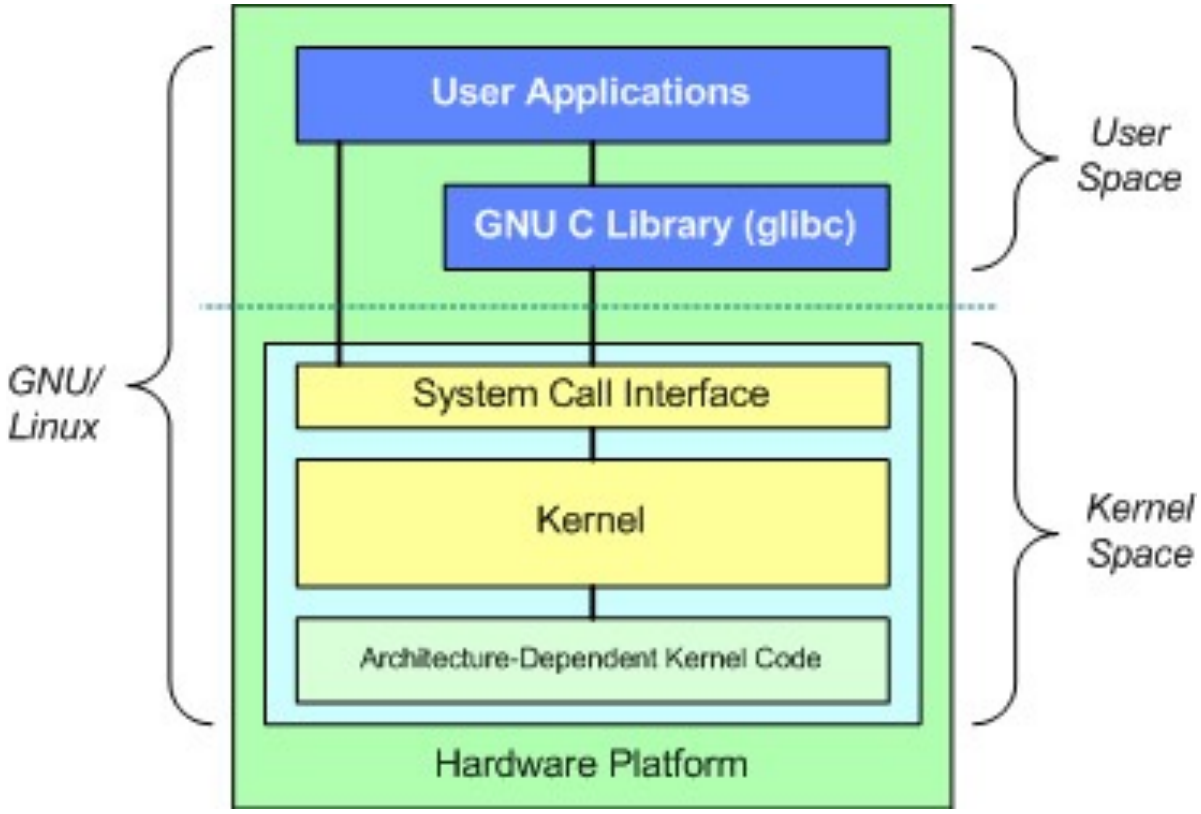
Hierarchical Style

- Hierarchical system organization
  - "Multi-layers of clients-servers".
  - Each layer exposes an interface (API) to be used by above layers.
- Each layer acts as a
  - Server to layers "above"
  - Client to layer(s) "below"
- Components are typically collections of procedures.
- Connectors are typically procedure calls under restricted visibility.
- Example: operating systems
- *Virtual machine* style results from fully opaque layers

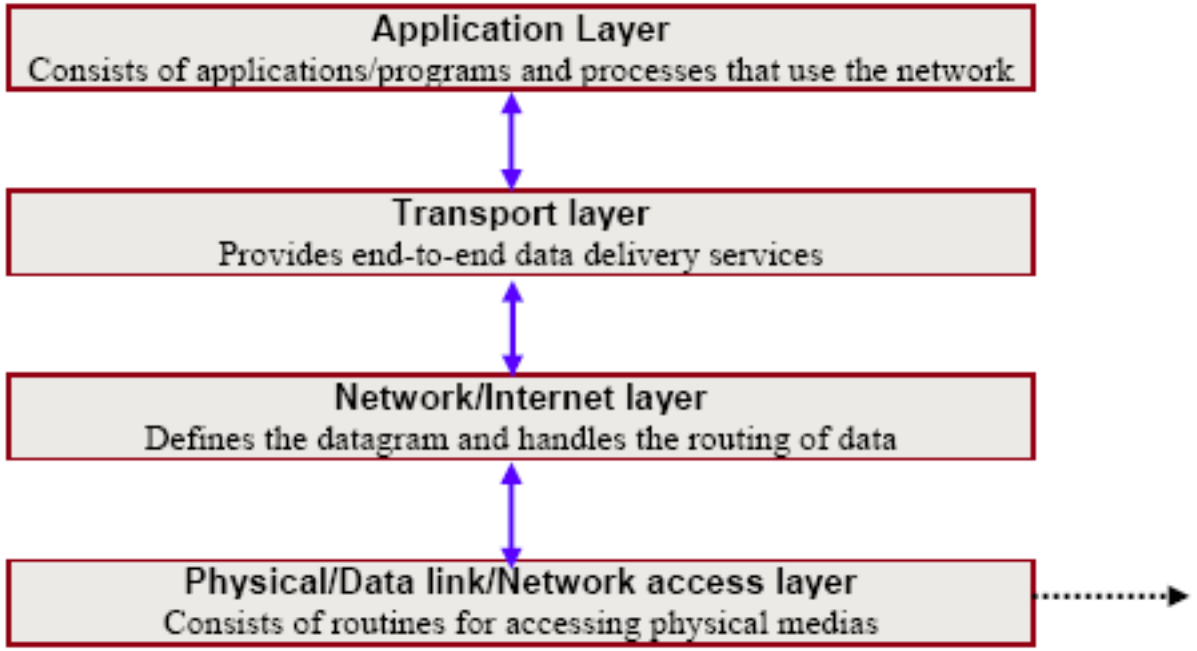
The Structure of an OS



The Structure of an OS

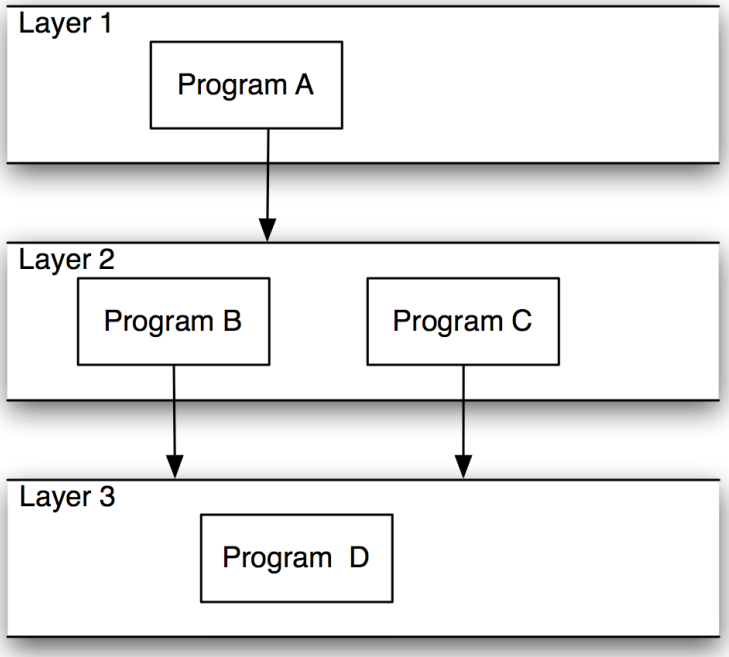


The Structure of TCP/IP

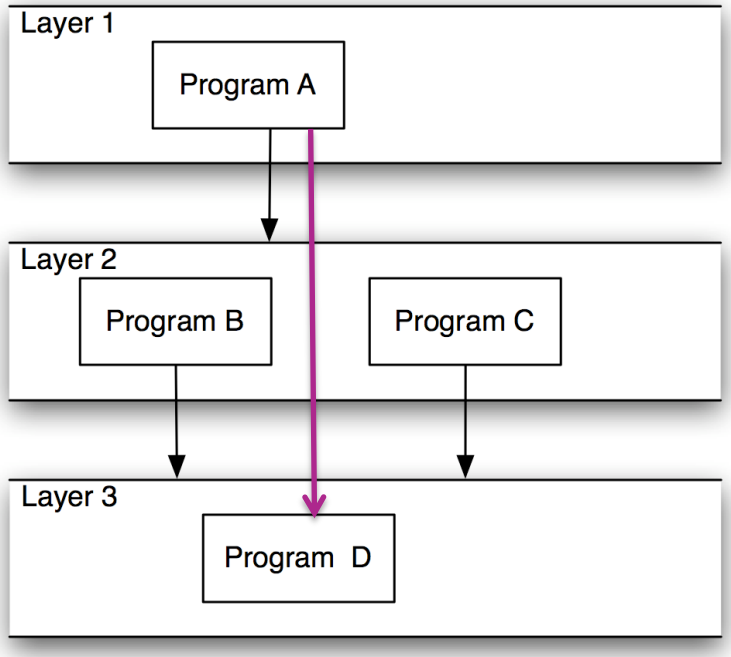


Non-strict Layering

Strict Layering



Non-strict Layering



- Practiced frequently.
- Breaks down the style, sometimes with undesirable consequences.

Hierarchical Style Advantages/Disadvantages

- Advantages
  - Clear dependence structure benefits evolution
    - Lower layers are independent from the upper layers
    - Upper layers can evolve independently from the lower layers as long as the interface semantics are preserved.
  - Reuse
    - Standardized layer interfaces for libraries and frameworks.
- Disadvantages
  - Not all systems are easily structured in layers.
  - Performance requirements may force the coupling of high-level functions to their lower-level implementations.

- Suitable for applications that involve distinct classes of services that can be organized hierarchically.