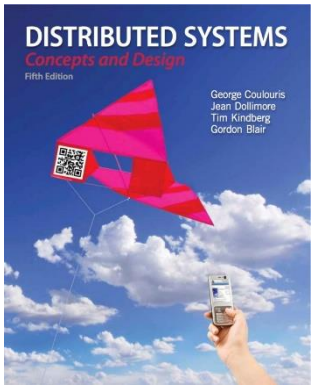
Operating System Architecture and Distributed Systems



Some concepts are drawn from Chapter 7

Dr. Minxian Xu
Associate Professor
Research Center for Cloud Computing
Shenzhen Institute of Advanced Technology, CAS
http://www.minxianxu.info/dcp

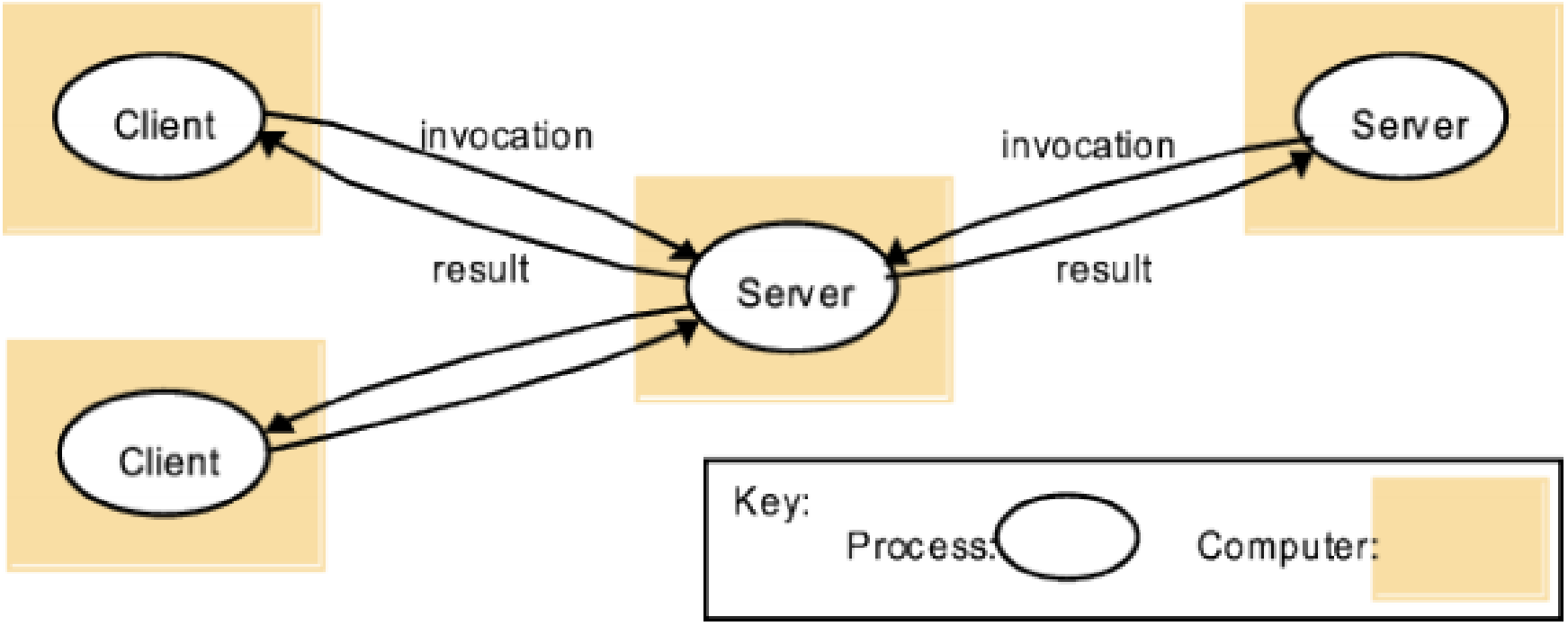
泰山不让土壤，故能成其大；河海不择细流，故能就其深。
——（秦）李斯

Review

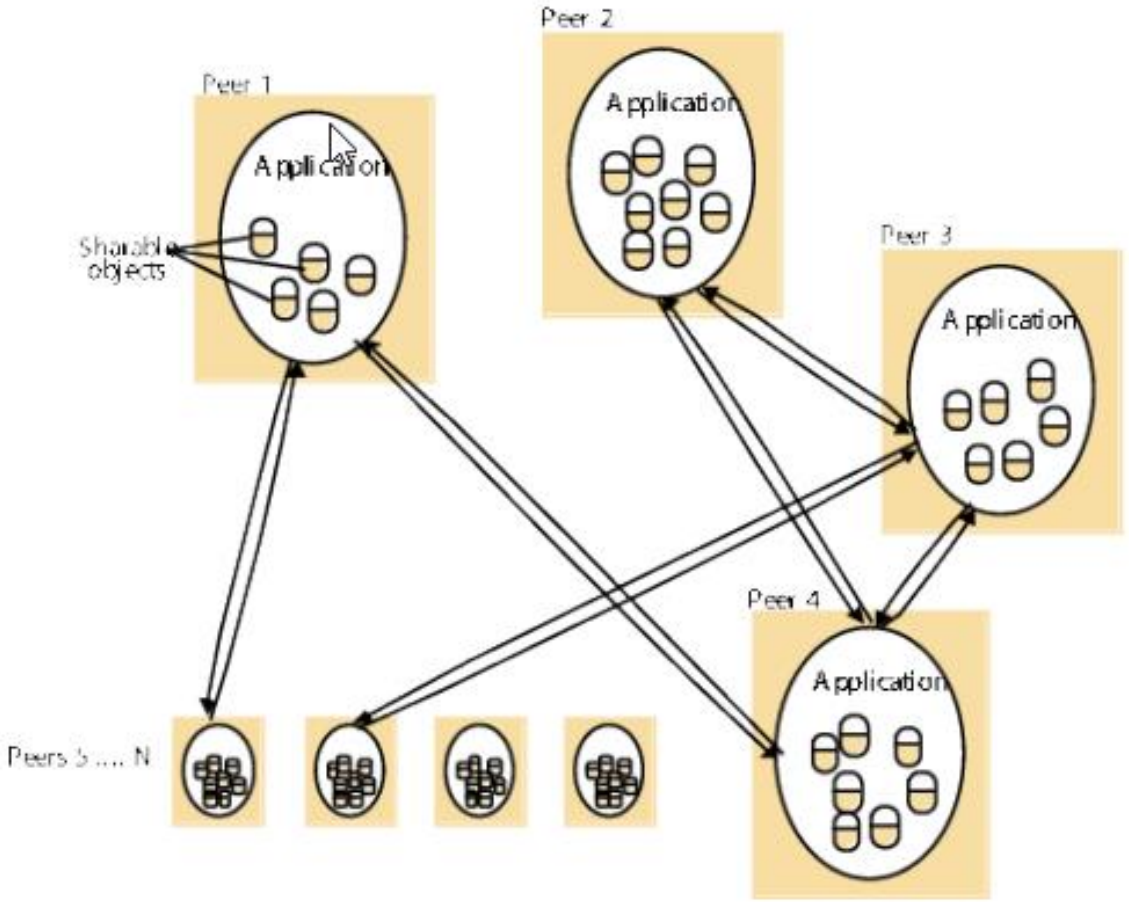
- Q1. Briefly explain the difference between a client-server architecture and a peer-to-peer architecture.

Review

Client-server



Peer-to-Peer


Peer 1
Application
Shareable objects
Peer 2
Application
Peer 3
Application
Peer 4
Application
Peers 5 .... N

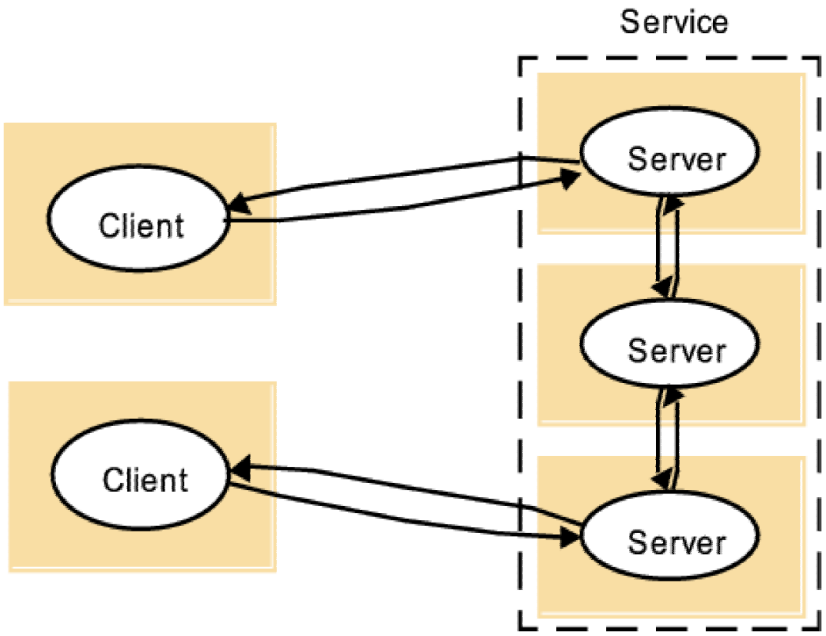
Review

- Q2. Briefly explain each of the following distributed system architecture variations, giving also a reason or a benefit for its use:

- Services provided by multiple servers
  - Proxy servers and caches
  - Mobile code and Mobile Agents
  - Network computers
  - Thin clients
- Tiered Architecture

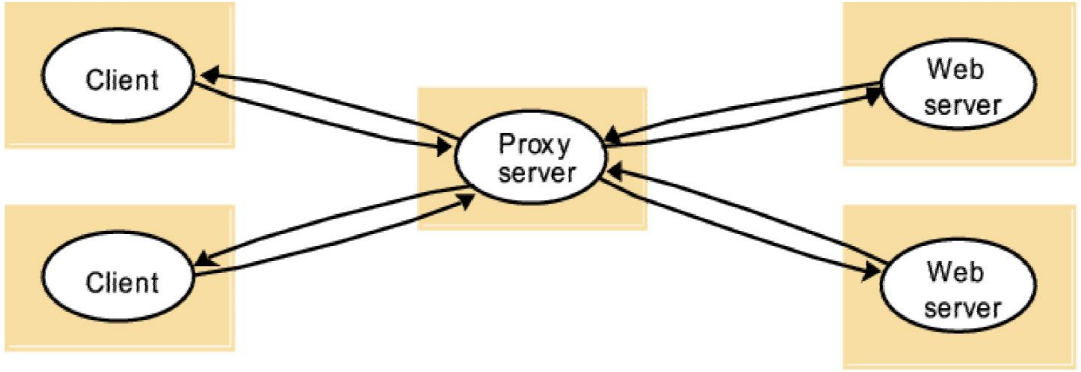
Services provided by multiple servers

- Services may be implemented as several server processes in separate host computers interacting as necessary to provide a service to client processes
- Servers may
  - Partition the set of objects on which the service is based and distribute those objects between themselves
  - Maintain replicated copies of them on several hosts
- Improve performance and reliability



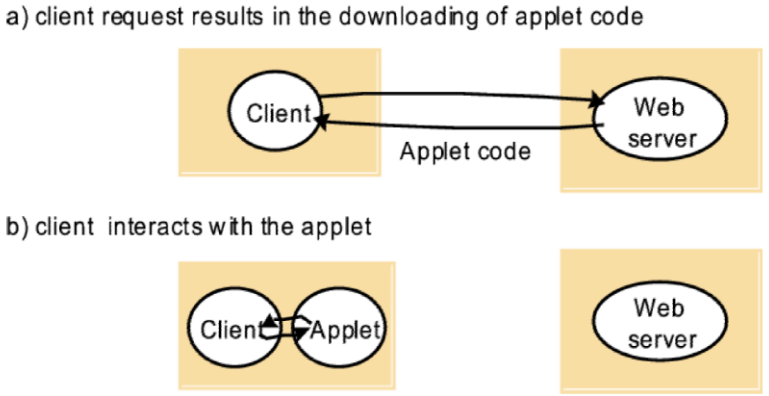
Proxy servers and caches

- Cache
  - A store of recently used data objects that is closer to the client
  - They may be co-located with each client or they may be located in a *proxy server* that can be shared by several clients
- Increase the availability and performance of the service by reducing the load on the wide area network and web servers
- Proxy servers can take on other roles --- better reliability
- Improved security
- Access restriction
- Privacy protection



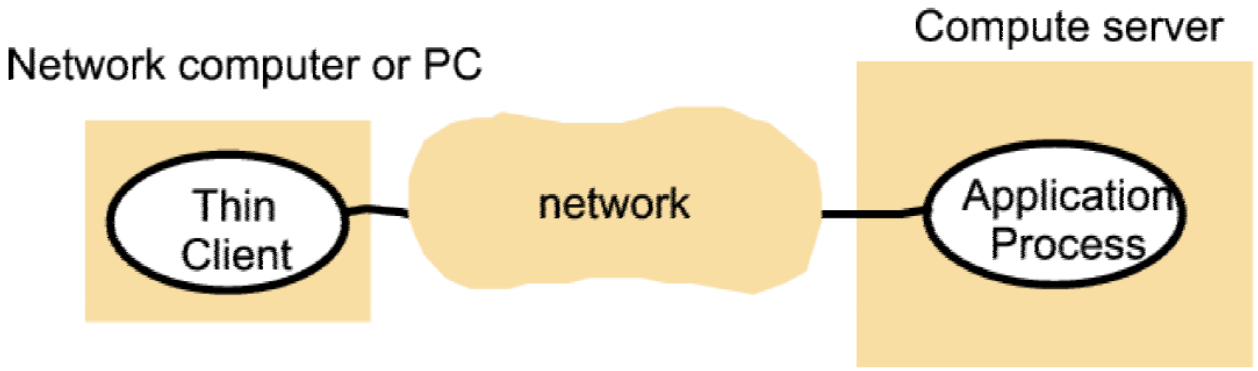
Mobile code and Mobile Agents

- Mobile code
  - Mobile Code is down loaded to the client and is executed on the client (e.g. applet).
  - Good interactive response
  - Security threat
- Mobile agents
  - Mobile agents are running programs that includes both code and data that travels from one computer to another.
  - They process data at the data source, rather than fetching it remotely
    - Less communication overhead by replacing remote invocations with local ones
  - Security threat



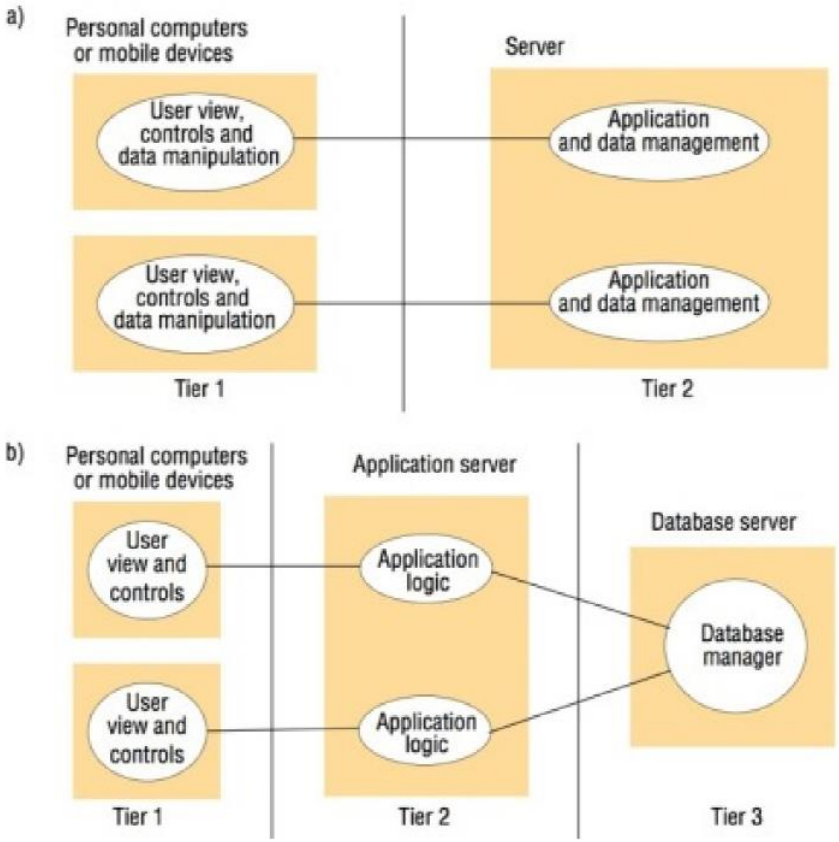
Network computers and thin clients

- Network Computers
  - Download their operating system and application software from a remote file system. Applications are run locally.
- Thin clients
  - Move complexity away from the end-user device
  - Local user interface, remote services or applications
  - Few assumptions or demands on the client device



Tiered Architecture

- Tiered architectures are complementary to layering, which deals with horizontal organization of services.
- Layering deals with vertical organization of services



Review

- Q3. Briefly explain the purpose of the following fundamental models and explain two important considerations for each:
- Interaction Model.
- Failure Model.
- Security Model.

Review

- Interaction Model
  - Models the interaction between processes of a distributed system - e.g. interaction between clients and servers or peers.
- Failure Model
  - Classifies the failures of processes and communication channels in a distributed system
- Security Model
  - Identifies the possible threats to processes and communication channels, as well as protecting encapsulated objects against unauthorized access.

Review

Q4. Explain the difference between a synchronous protocol and an asynchronous protocol.

Review

- Synchronous communication blocks on both send and receive operations.
  - When a send is issued the sending process is blocked until the receive is issued.
  - Whenever the receive is issued the process blocks until a message arrives.
- In Asynchronous communication the send is nonblocking.
  - The sending process returns as soon as the message is copied to a local buffer and the transmission of the message proceeds in parallel.
  - Receive operation can be blocking or non-blocking (non-blocking receives are not normally supported in today's systems).

Operating System Architecture and Distributed Systems (DS)

- Explore the architecture of a kernel suitable for a distributed system.
- A key principle of DS is openness and with this in mind, let us examine the major kernel architectures:
  - Monolithic kernels
  - Layered architecture-based kernels
  - Micro-kernels

Open DS and System Software

- A open DS should make it possible to:
  - Run only that ("specific" components of) system software at each computer that is necessary for its particular role in the system architecture.
    - For example, system software needs of laptops and dedicated servers are different and loading redundant modules wastes memory resources.
  - Allow the software implementing any particular service to be changed independent of other facilities.
  - Allow for alternatives of the same service to be provided, when this is required to suit different users or applications.
  - Introduce new services without harming the integrity of existing ones.

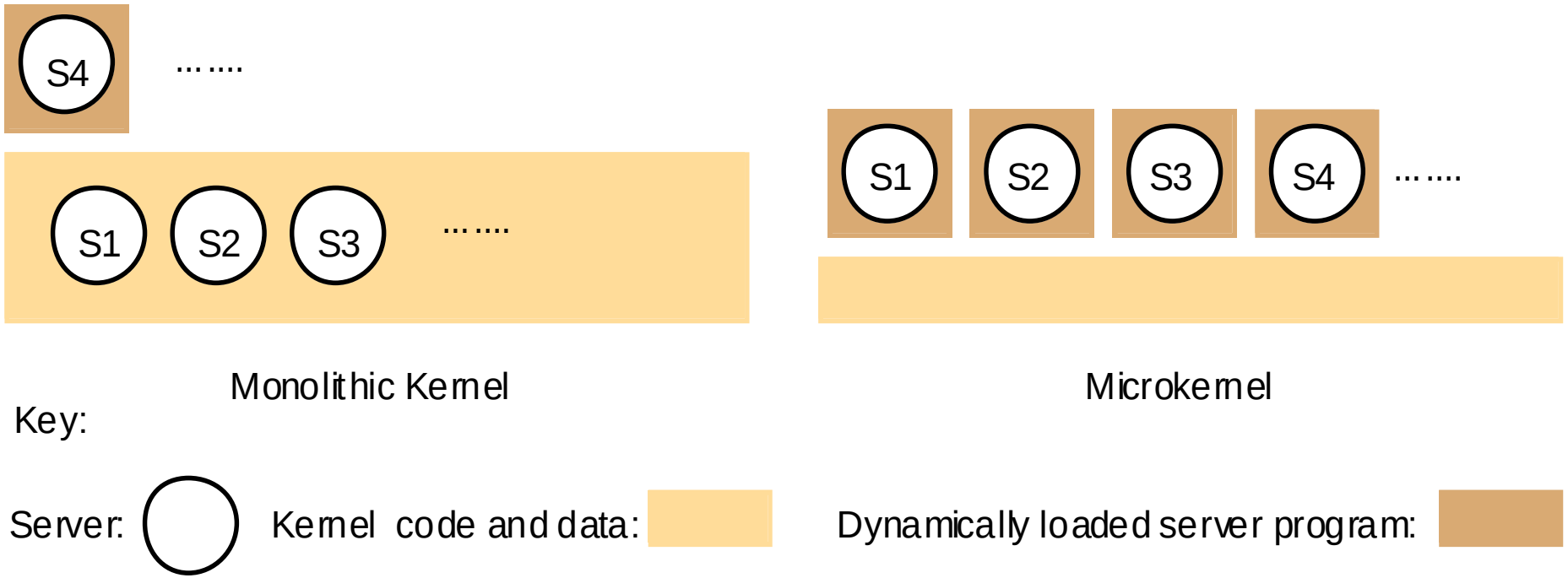
Separating Mechanisms and Policies in OS and DS

- A Guiding Principle of OS design:
  - The separation of fixed resource management "mechanisms" from resource management "policies", which vary from application to application and service to service.
  - For example, an ideal scheduling system would provide mechanisms that enable a multimedia application such as videoconferencing to meet its real-time demands while coexisting with a non-real-time application such as web browsing.
- That is kernel would provide only the most basic mechanisms upon which the general resource management tasks at a node are carried out.
- Server modules would be dynamically loaded as required, to implement the required RM policies for the currently running applications.

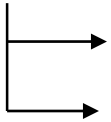
OS/Kernel Architecture

- The two key examples of kernel design approaches are:
  - Monolithic
  - Microkernel
- These two designs differ primarily in the decision as to what functionality belongs in the kernel and what is left to server processes that can be dynamically loaded to run on top of it.
- In literature, we find predominantly 3 types of OSs:
  - Monolithic OS
  - Layered OS
  - Microkernel-based OS

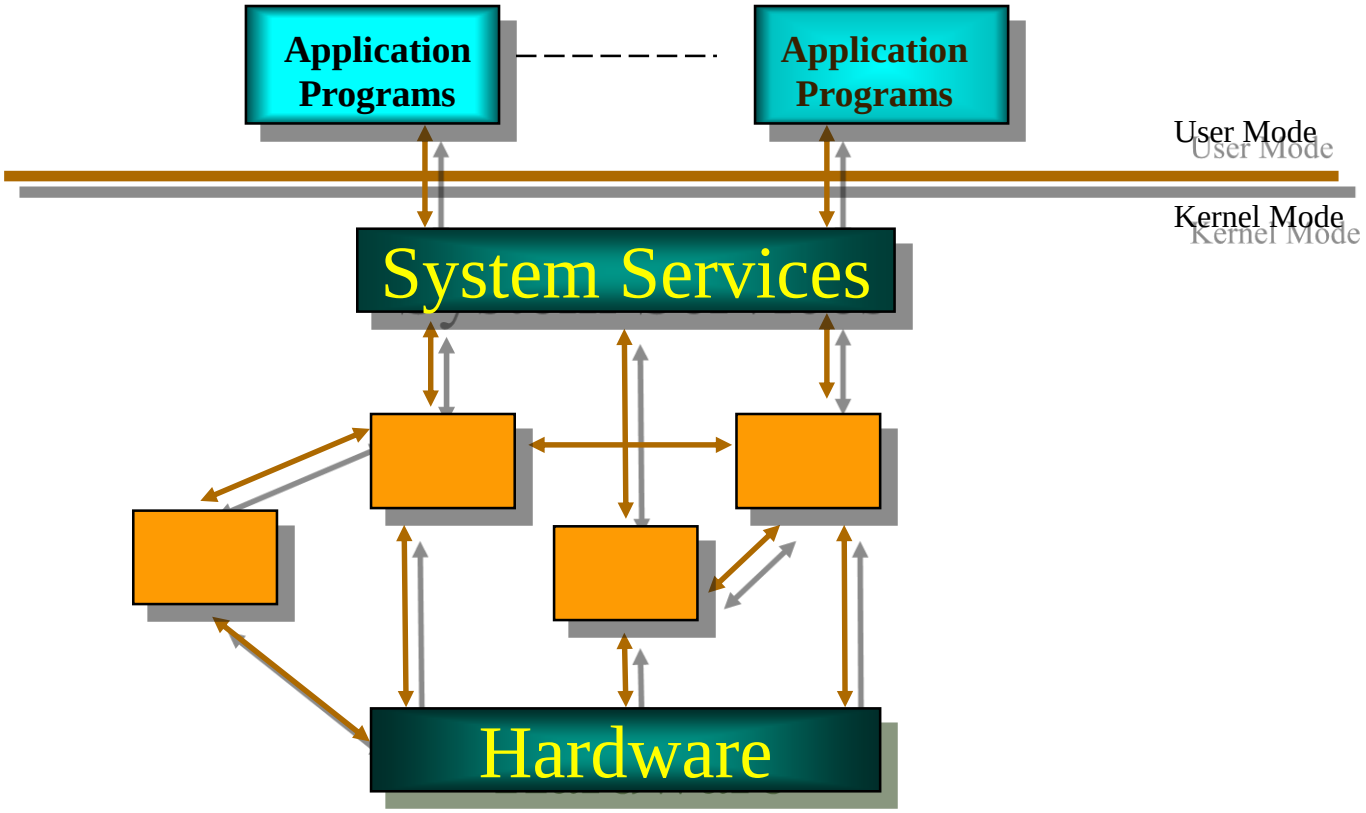
Monolithic kernel and microkernel



Operating System Models

- Serve as frameworks that unify capabilities, services and tasks to be performed
- Three approaches to building OS....
  - Monolithic OS
  - Layered OS
  - Microkernel based OS
- Client server OS
-  Suitable for distributed systems
- Simplicity, flexibility, and high performance are crucial for OS.

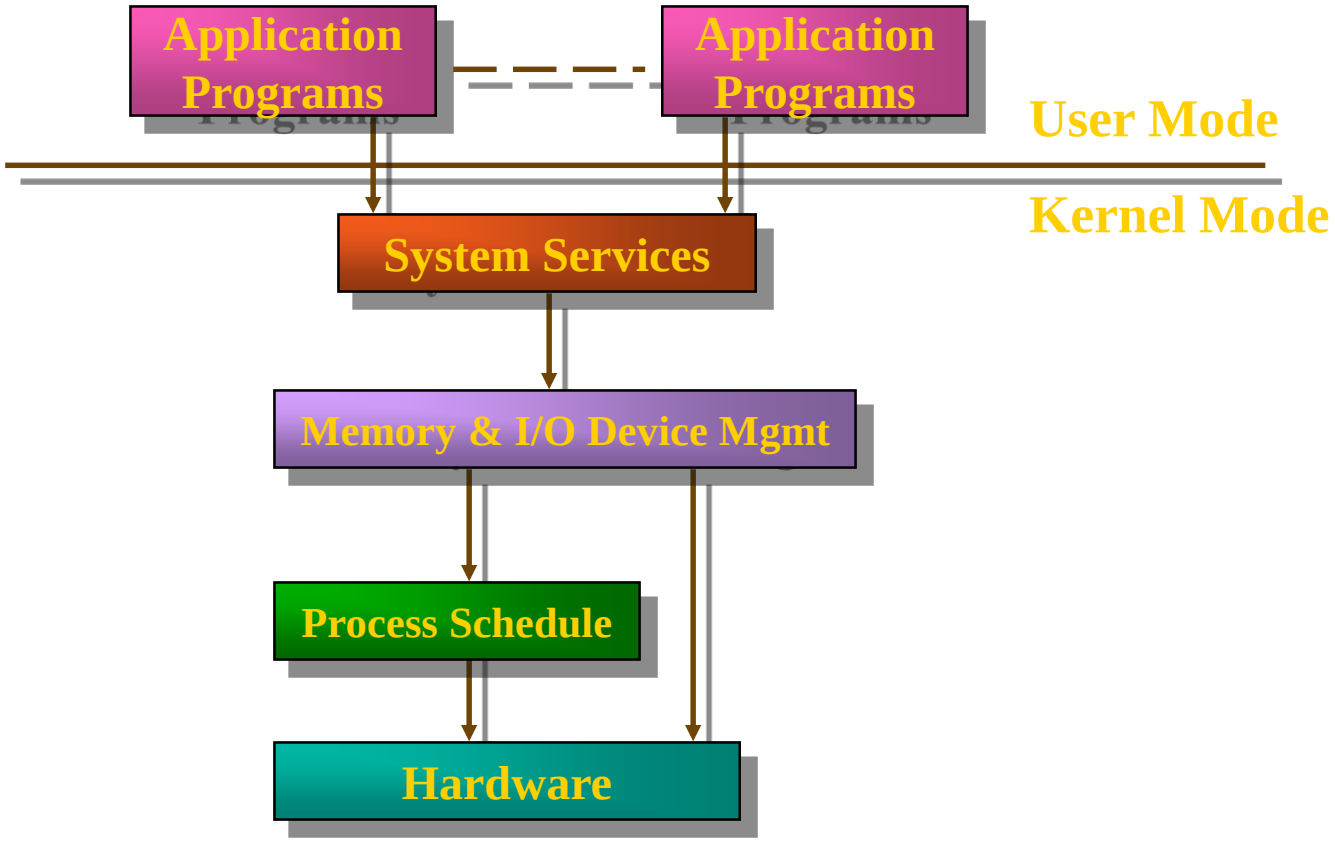
Monolithic Operating System



- Better application Performance
- Difficult to extend

Ex: MS-DOS

Layered OS



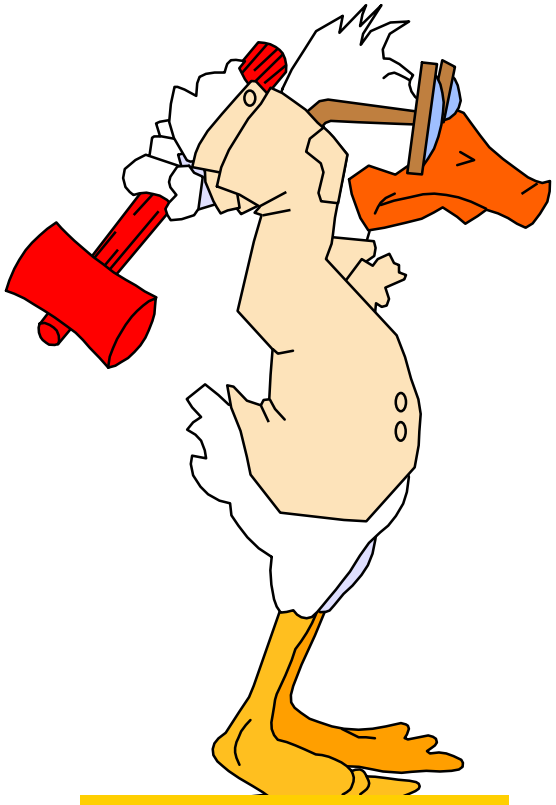
- Easier to enhance
- Each layer of code access lower level interface
- Low-application performance

Ex : UNIX

Traditional OS

User Mode

Kernel Mode



OS Designer

Disadvantages of Monolithic OS

- It is massive:
  - It performs all basic OS functions and takes up in the order of megabytes of code and data
- It is undifferentiated:
  - It is coded in a non-modular way (traditionally) although modern ones are much more layered.
- It is intractable:
  - Altering any individual software component to adapt it to changing requirements is difficult.

New trend in OS design: Separating mechanisms and policies

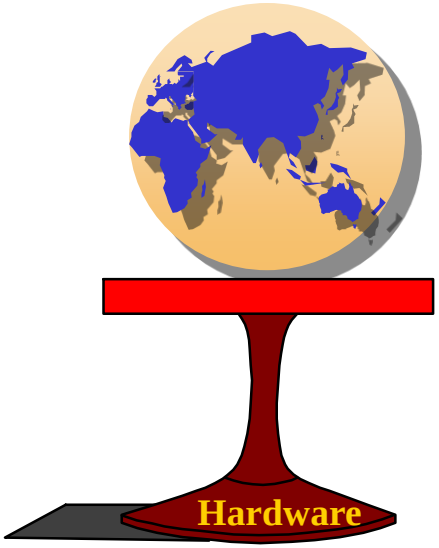
Application Programs

Servers

Application Programs

User Mode

Kernel Mode

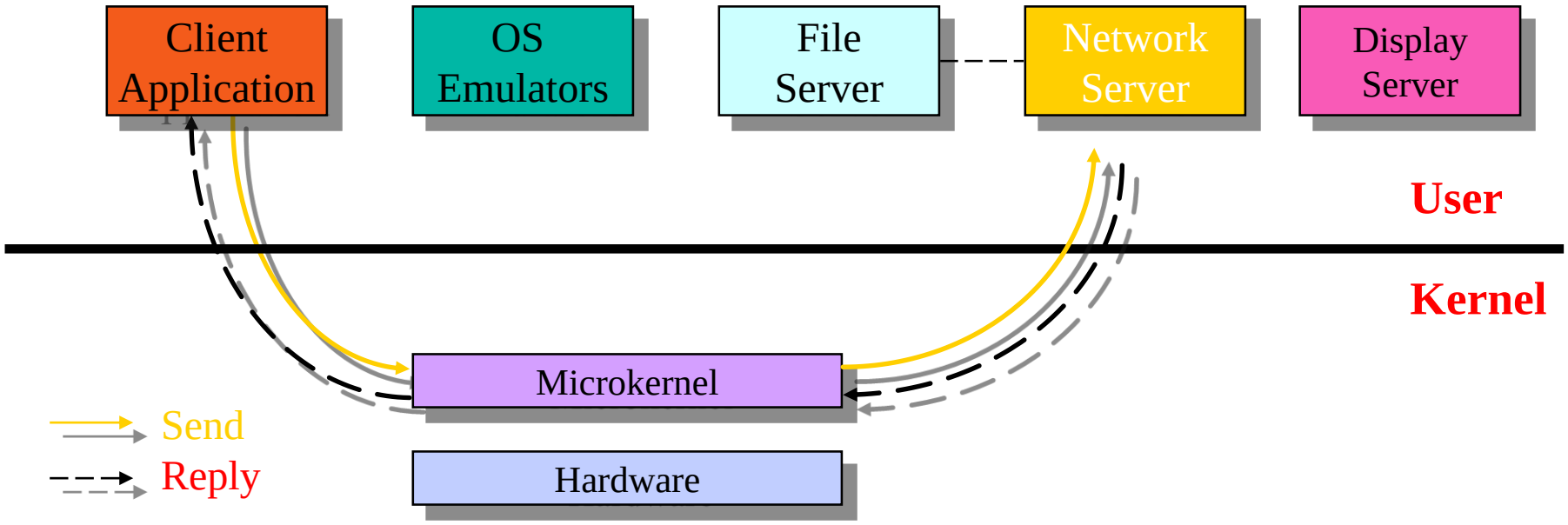


Microkernel
(very basic functions)

Micro-kernel

- Compared to monolithic, microkernel design provides only the most basic abstractions,
  - address space, threads and local IPC.
- All other system services are provided by servers that are dynamically loaded precisely on those computers in the DS that require them.
- Clients access these system services using the kernel's message-based invocation mechanisms.

Microkernel/Client Server OS

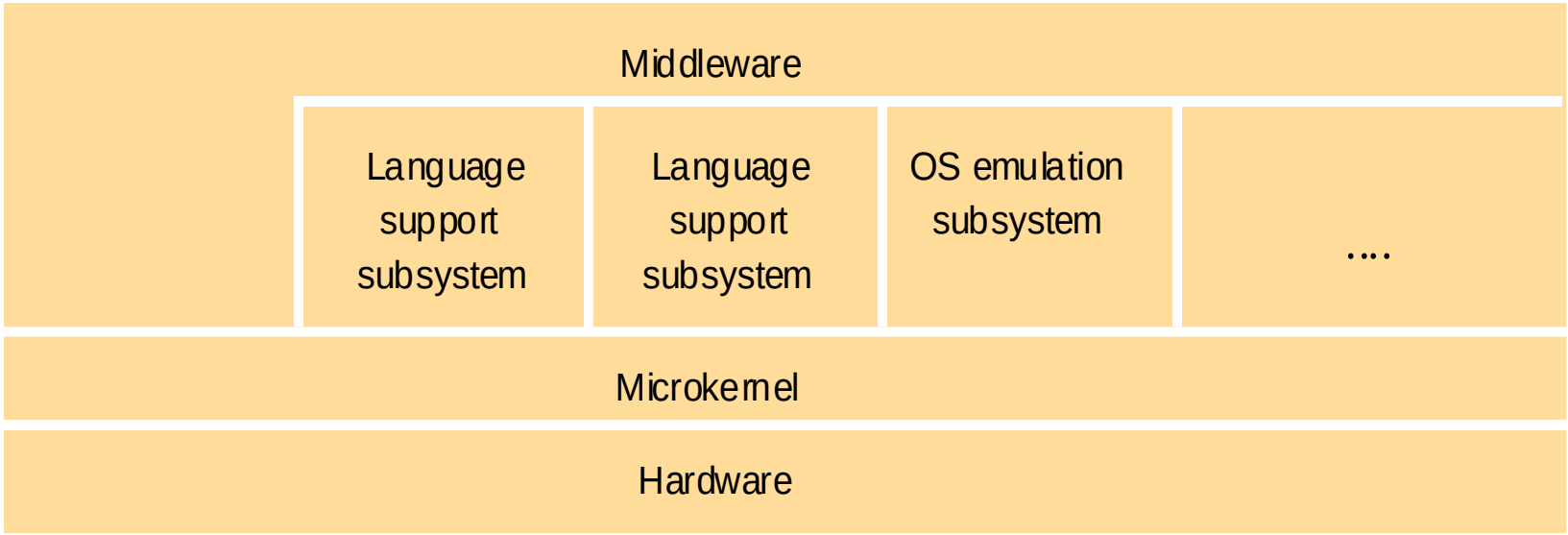


- Tiny OS kernel providing basic primitive (process, memory, IPC)
- Traditional services becomes subsystems
- OS = Microkernel + User Subsystems

Ex: Mach, QNX, Windows NT!

The role of the microkernel (MK)

- MK appears as a layer between H/W and a layer of major system components (subsystems). If performance, rather than portability is goal, then middleware may use facilities of MK directly.



The microkernel supports middleware via subsystems

The microkernel story is full of good ideas and blind alleys (Liedtke)

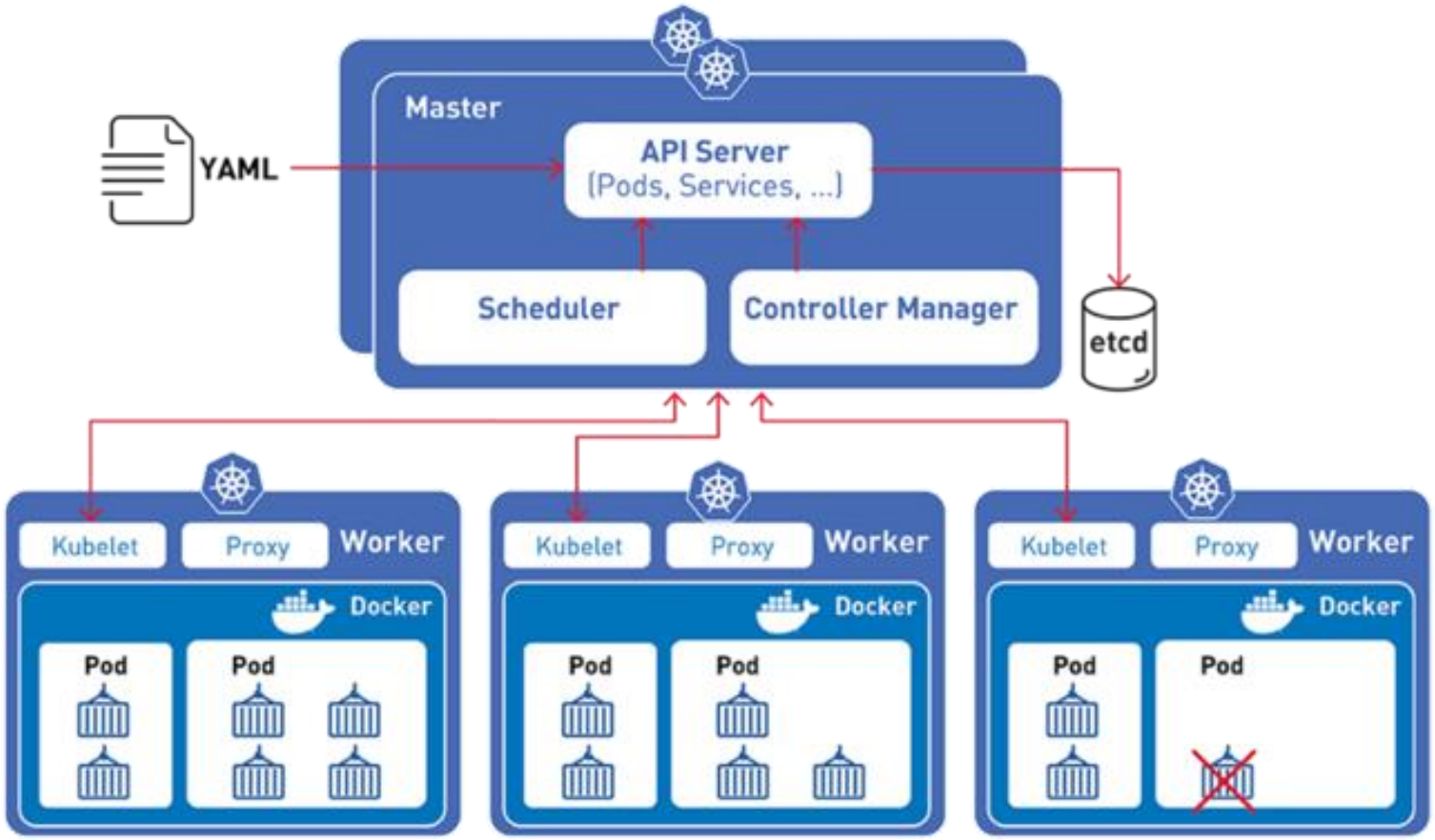
- Although the concept of microkernel is beautiful, the reality is very cruel:




Few Popular Microkernel Systems

- MACH, CMU (Carnegie Mellon University)
  - supports OS emulators such as Unix and OS/2.
- PARAS (C-DAC, India) for PARAM Supercomputers
- ChorusOS (Sun, USA) Realtime OS (RTOS)
- seL4 (created by NICTA/Data61, Australia)
- QNX - Unix-like RTOS (Canada, BlackBerry)
  - used in a variety of devices including cars and mobile phones (e.g., BlackBerry).
  - Intel x86, MIPS, PowerPC, StrongARM..
- Windows NT – original design
- HarmonyOS (Huawei)

Kubernetes: distributed operating system with Microkernel



Huawei Harmony OS

- "HarmonyOS is completely different from Android and iOS. It is a microkernel-based, distributed OS that delivers a smooth experience across all scenarios. It has a trustworthy and secure architecture, and it supports seamless collaboration across devices. You can develop your apps once, then flexibly deploy them across a range of different devices."

Comparison: Monolithic and Micro-kernel OS Design

- The main advantages of a MK-based OS:
  - A relative small kernel is more likely to be free of bugs than one that is larger and complex.
  - Extensibility and its ability to enforce modularity behind memory protection boundaries
- The advantage of a monolithic OS:
  - Relative efficiency with which operations can be invoked is high because even invocation to a separate user-level address space on the same node is more costly.

Hybrid Approaches

- Many modern OSs follow hybrid approach in OS structure. E.g., Windows NT.
- Pure microkernel OSs such as Chorus & Mach have changed over time to allow servers to be loaded dynamically into the kernel address space or into a user-level address space.
- Some OSs (such as SPIN) use event-based model as a mechanism for interaction between modules grafted into the kernel address space.

Summary

- OSs provide various types of facilities/services to support middleware for distributed system:
  - encapsulation, protection, and concurrent access and management of node resources.
- Three types of OS:
  - Monolithic OS
  - Layered OS
  - Microkernel-based OS
- New OS designs provide flexibility in terms of separating mechanisms from policies.
- Microkernel based systems are flexible
  - Quite popular model for OS design for embedded systems
  - New Emerging optimized Kernels like *nanokernel* or *picokernel*

Additional References

- Rajkumar Buyya, The Design of PARAS Microkernel, Centre for Development of Advanced Computing (C-DAC), 1998.
  - http://www.buyya.com/microkernel/chap2.pdf
- Gernot Heiser, Gerwin Klein, June Andronick, *seL4 in Australia: From Research to Real-World Trustworthy Systems*, Communications of the ACM, April 2020.
  - https://cacm.acm.org/magazines/2020/4/243641-sel4-in-australia/fulltext

- Demo
- JSON Client Server
- JSON Simple Domo

JSON Example

- "JSON" stands for "JavaScript Object Notation"
  - Despite the name, JSON is a (mostly) language-independent way of specifying objects as name-value pairs
- Example
-
```
{"skillz": {
    "web":[
        { "name": "html",
          "years": 5
        },
        { "name": "css",
          "years": 3
        }]
    "database":[
        { "name": "sql",
          "years": 7
        }]
}}
```

JSON Syntax

- An *object* is an unordered set of name/value pairs
  - The pairs are enclosed within braces, { }
  - There is a colon between the name and the value
  - Pairs are separated by commas
  - Example: { "name": "html", "years": 5 }

JSON Syntax

- A *value* can be: A string, a number, true, false, null, an object, or an array
  - Values can be nested
- *Strings* are enclosed in double quotes, and can contain the usual assortment of escaped characters
- *Numbers* have the usual C/C++/Java syntax, including exponential (E) notation
  - All numbers are decimal--no octal or hexadecimal
- *Whitespace* can be used between any pair of tokens