Lecture 7

Architectural Patterns:
Layers, Façade, Microkernel,
Design Pattern: Strategy

Layers Pattern

Operative Metaphor:
Assembly Line

Layers

- Architectural layers are collaborating sections of an overall complex system that provide several benefits such as:
  - supporting incremental coding and testing, allowing localization of changes
  - well-defined interfaces allow substitution of different layers
  - protection between collaborating layers
  - Layers support a responsibility-driven architecture that divides subtasks into groups of related responsibilities

Layers Pattern

- In the pure sense, each layer provides services to the layer *directly* above it, and acts as a client to the layer *directly* below it
- In an "impure" implementation, distanced layers can be "bridged" which allows communication between them but reduces portability and flexibility and plug and play capability
- Each layer provides a defined interface to the layers above and below it
- Higher layers provide increasing levels of abstraction

Features

- Often, layered architectures are applied to virtual machine architectures (such as in interpreters), in that hardware layers are virtualized in software, and either act as mediators to actual hardware layers, or are stubbed out entirely
- A layered system can be seen as a static pipes and filters system but without the pipes, where the filters talk directly to their neighbors

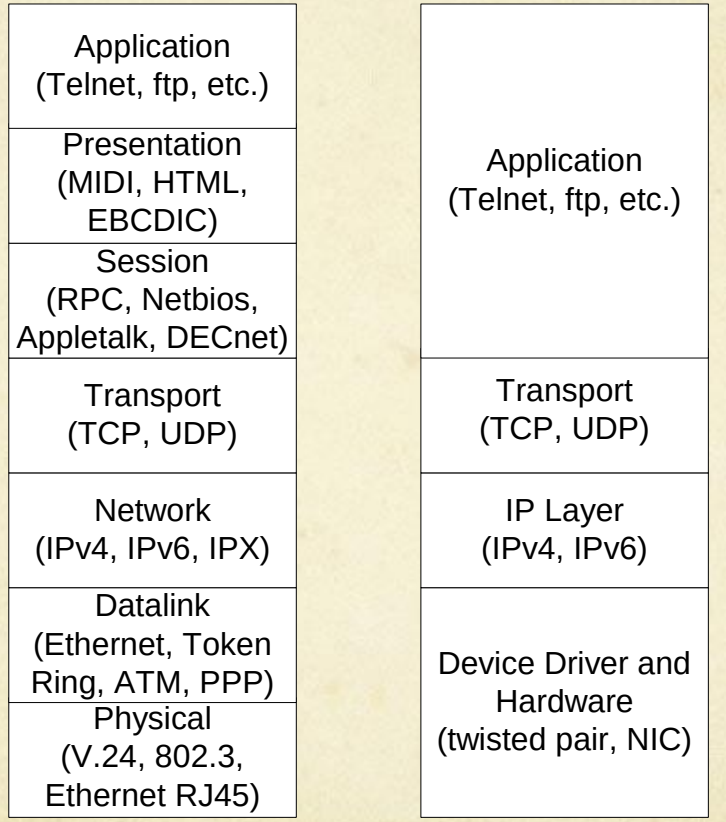
Benefits

- A layered pattern supports increasing levels of abstraction, thus simplifying design
- This allows a complex problem to be *partitioned* into a sequence of manageable incremental strategies (as layers)
- Like Pipes and Filters, layers are loosely coupled, so maintenance is enhanced because new layers can be added affecting only two existing components (as layers)
- Layers support plug-and-play designs. As long as the interfaces do not change, one layer can be substituted for another changing the behavior of the layer system
- Lowest Common Denominator issues of Pipes and Filters is not present in Layers

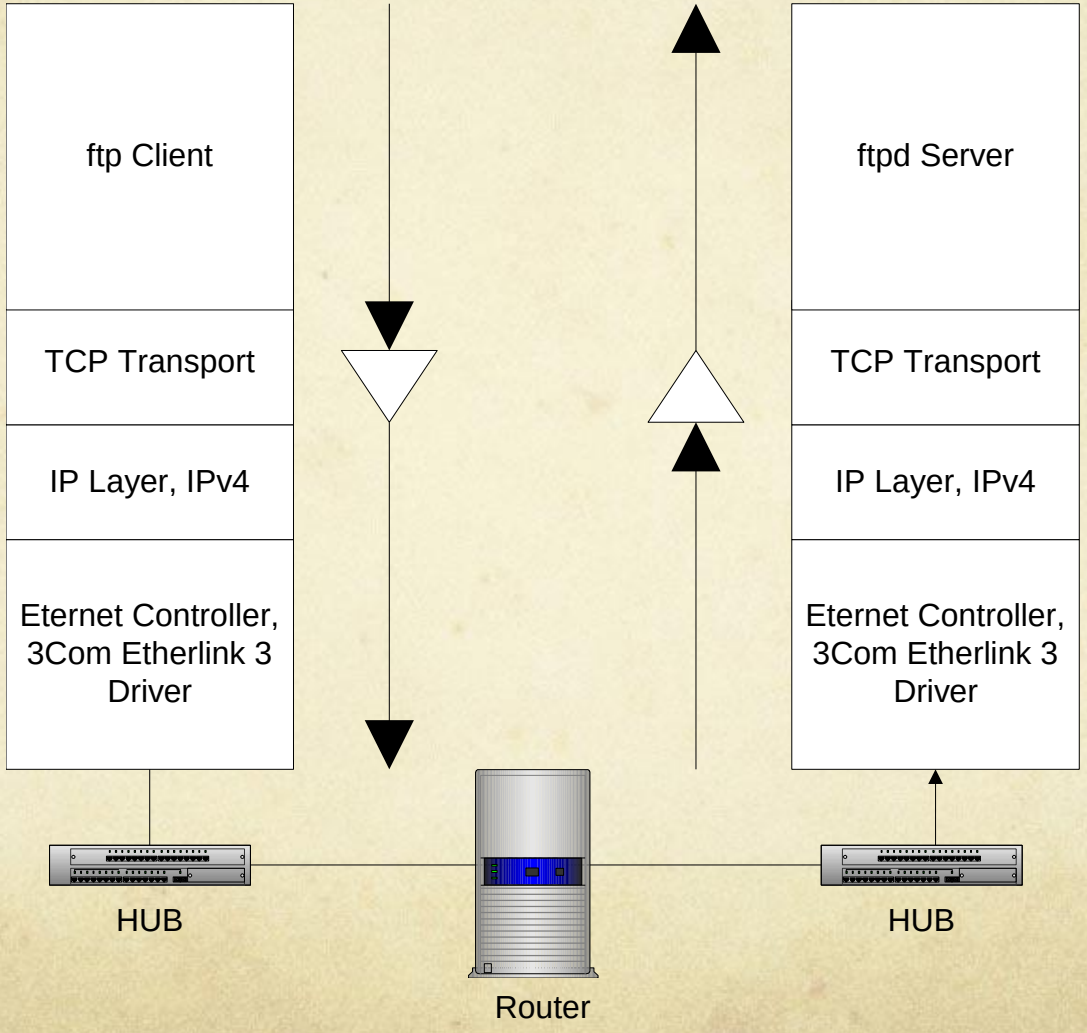
Disadvantages

- Close coupling of juxtaposed layers lowers maintainability
- Each layer must manage all data marshaling and buffering
- Lower runtime efficiency
- Sometimes difficult to establish the granularity of the various layers (10 layers or 4?)

The Usual Suspect: TCP/IP Protocol Stack

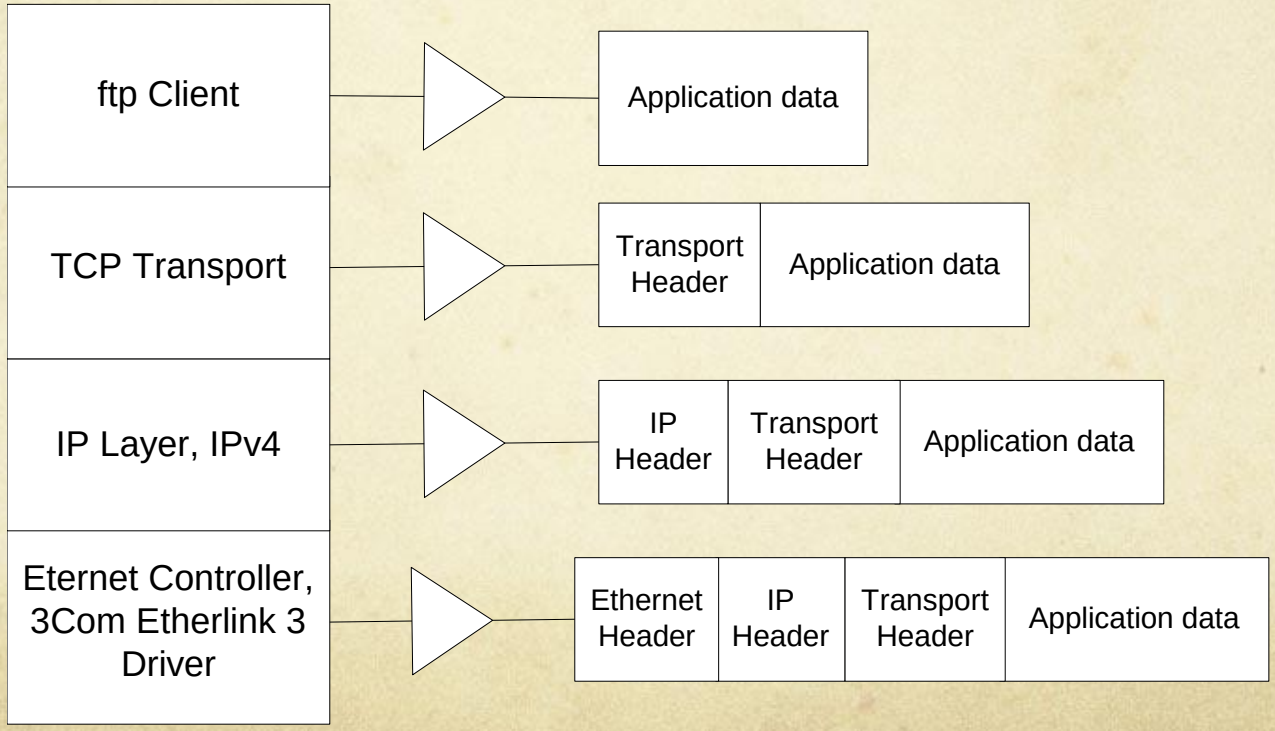


Protocol Communication



Data Encapsulation

- Application puts data out through a socket
- Each successive layer wraps the received data with its own header:



The IP Layer

- The IP layer allows packets to be sent over gateways to machines not on the physical network
- Addresses used are IP addresses, 32-bit numbers divided into a network address (used for routing) and a host address
- The IP protocol is connectionless, implying:
  - gateways route discrete packets independently and irrespective of other packets
  - packets from one host to another may be routed differently (and may arrive at different times)
  - non-guaranteed delivery

The Transport Layer

- Unix has two common transports
  - User Datagram Protocol
    - record protocol
    - connectionless, broadcast
    - *Metaphor*: Postal Service
  - Transmission Control Protocol
    - byte stream protocol
    - direct connection-oriented
    - *Metaphor*: Phone Service circa 1945
      - "Sarah, this is Andy, get me Barney please."

Façade

Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

Operative Metaphor: Telephone

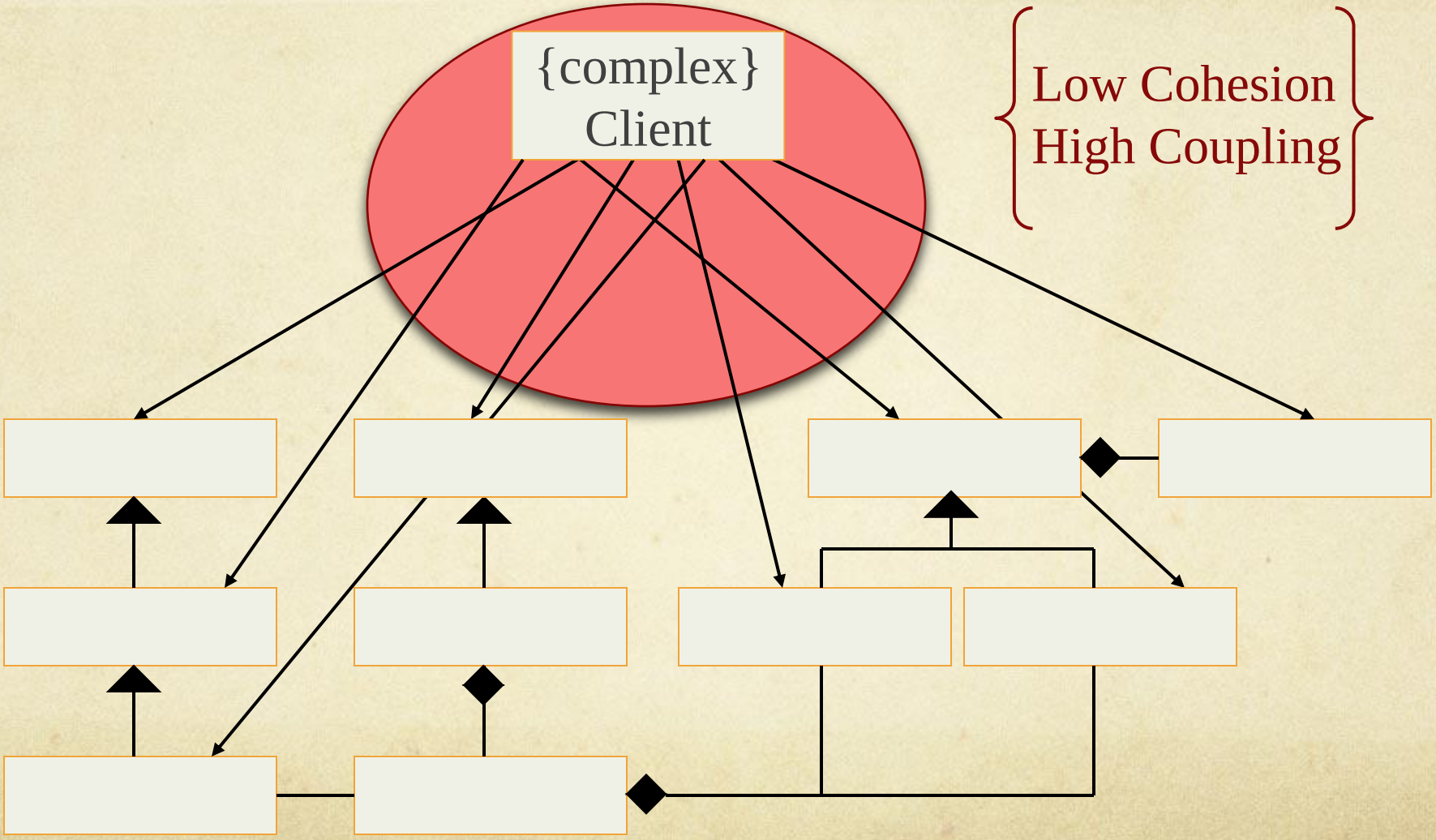
Characteristics

- Use the Façade pattern when you want to hide the complexity of a rich subsystem behind a simple interface
- A Façade reduces the number of classes a client has to deal with, thus reducing the dependencies between classes (which reduces complexity)

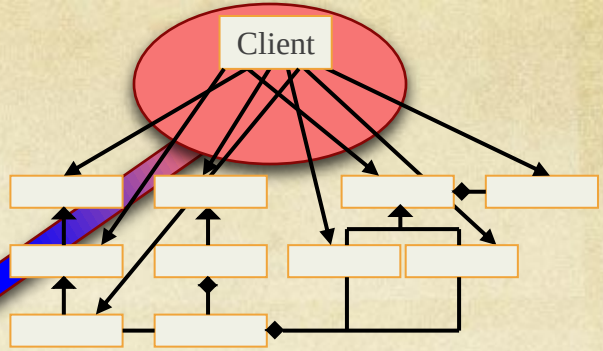
Intent

- Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- When a class is defined that provides a common interface to a disparate set of interfaces… The disparate interfaces may be to a set of functions, a framework, a group of other classes, or a subset (local or remote).

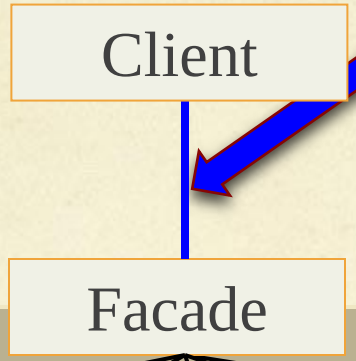
Motivation

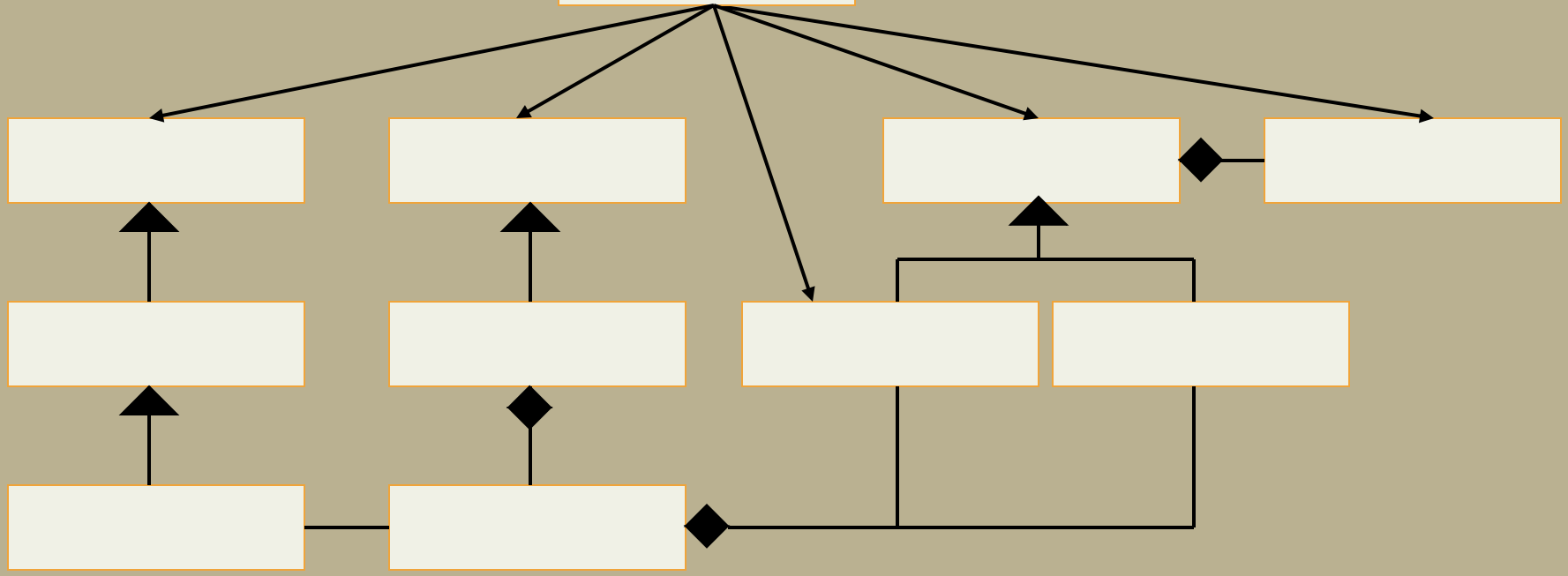


Motivation



High Cohesion
Low Coupling





Motivation

- Reduce complexity for the client.
- Does not prevent client from using subsystem.
- Promotes loose coupling between the client and the subsystem.
- Assists with defining layers (entry points) for the client to the subsystem.

Benefits

- Flexibility is still achieved in that most façades can be bypassed when necessary (sometimes called “transparent” façades)
- Minimizes the number of interdependencies between layers, promoting subsystem independence and substitution
- Promotes loose coupling between layers, allowing subsystem components to change without affecting clients of the façade

Consequences

- Limits the clients ability to use features of the subsystem.
- Complicates communication between the client and the subsystem. An example would be propagation of the source on an error from within the subsystem…is the error going to be meaningful to the “distant” client?

Strategy

A Strategy defines a family of encapsulated algorithms, and lets them vary interchangeably independent of the the clients that use them

Motivation

- Anytime you have multiple methods of accomplishing a task, for example, multiple ways to encrypt, you need to be able to switch your method of encryption, and this is traditionally done in some form of "if" or "switch" statement
- This is messy because it hard-codes the available algorithmic solutions into a "manager" clause that makes maintenance and alteration difficult

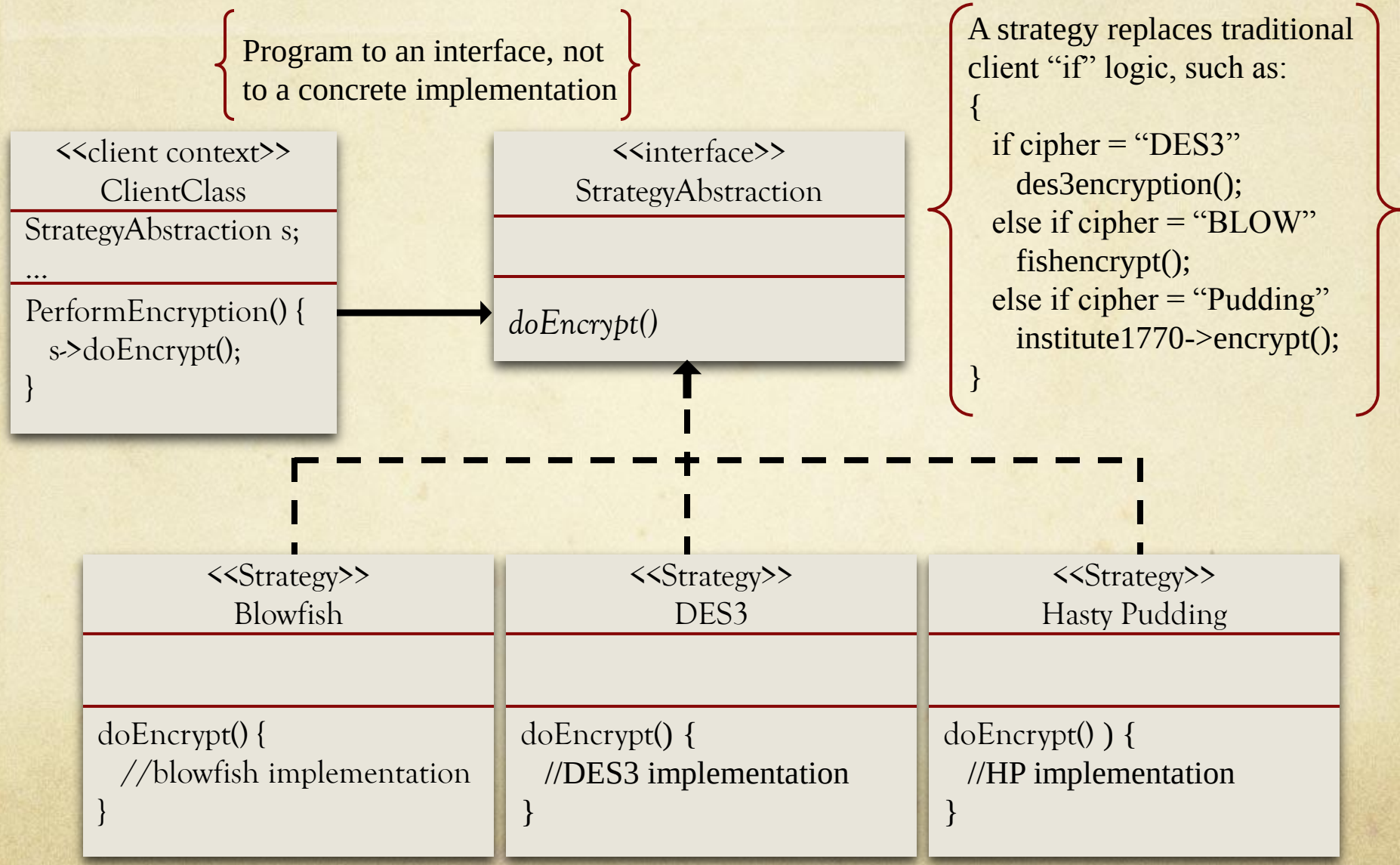
```
{
  if cipher = "DES3"
    des3encryption();
  else if cipher = "BLOW"
    fishencrypt();
  else if cipher = "HP1770"
    hp1770->encrypt();
}
```

Client code…

The Solution

- Define an abstract interface that just defines the interface for the various methods of doing something, effectively replacing the client's "if" logic. This is the abstract interface of a *Strategy*.
- Define an abstract method that is generic on the interface, such as "doEncrypt()". This is the abstract strategy method.
- Define a concrete classes that encapsulate the various methods by which something can be done (the various strategy implementations).
- Implement each independent strategy in its own concrete class, thus making each implementation a *type of Strategy*.

UML


Program to an interface, not to a concrete implementation
<<client context>>
ClientClass
StrategyAbstraction s;
...
PerformEncryption() {
s->doEncrypt();
}
<<interface>>
StrategyAbstraction
doEncrypt()
A strategy replaces traditional client "if" logic, such as:
{
if cipher = "DES3"
des3encryption();
else if cipher = "BLOW"
fishencrypt();
else if cipher = "Pudding"
institute1770->encrypt();
}
<<Strategy>>
Blowfish
doEncrypt() {
//blowfish implementation
}
<<Strategy>>
DES3
doEncrypt() {
//DES3 implementation
}
<<Strategy>>
Hasty Pudding
doEncrypt()) {
//HP implementation
}

Examples

- Ruby (1 and 2)
- C#

Microkernel Pattern

"Perfection is not achieved when there is nothing left to add, but when there is nothing left to take away"
--Antoine de St. Exupery

Microkernel

- The Microkernel pattern applies the principles of modular and layered development to the operating system (or virtual machine) itself
- Modules constitute dynamic Layers
- As such, this pattern promotes flexible architectures which allow systems to adapt successfully to changing system requirements and interfaces

Definitions

- A Microkernel is a highly Spartan modular subsystem composed of OS-neutral abstractions, providing only essential services such as process abstractions, threads, IPC, and memory management primitives.
- All device drivers, etc., which are normally part of an OS kernel, run on the microkernel as just another user process
- Multiple operating systems can then be layered *on top of* these abstractions, and are thus viewed as simply another *application*.
- This focus on modularity allows for scalability, extensibility and portability not found in monolithic operating systems (Unix, Linux, DOS, etc.)

Features

- Because the microkernel provides only rudimentary core facilities, different OS personalities (such as BSD Unix, Linux, NT, etc.) can be hosted on the microkernel.
- Because of its highly modular nature, many of the services commonly found in “kernel space” are found in “user space” on a microkernel

Features

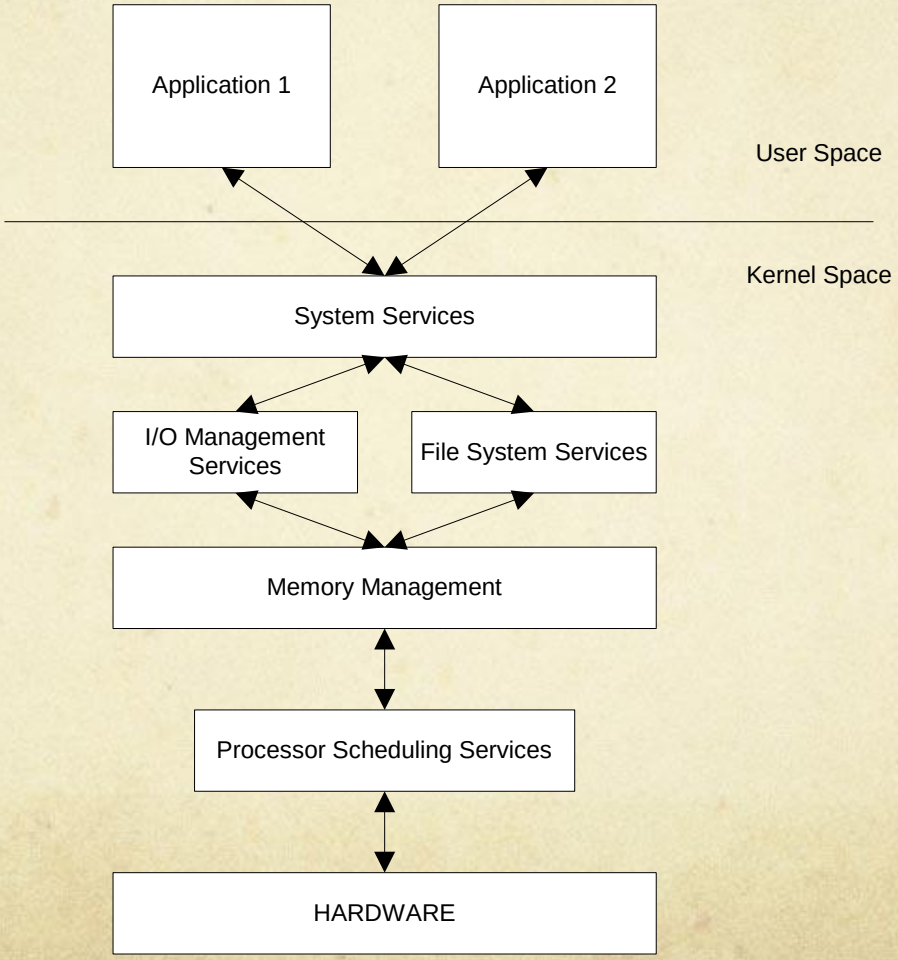
- Flexibility (can restart modules without rebooting the OS)
- Lower fixed memory demand: The L4 (Mach) Microkernel only takes up about 32 *Kilo*bytes of memory.
- However, a microkernel + regular OS will probably take up *more* memory than a simple OS would take up, because of the additional memory required by the microkernel itself
- SMP delivery is easier

Strategy

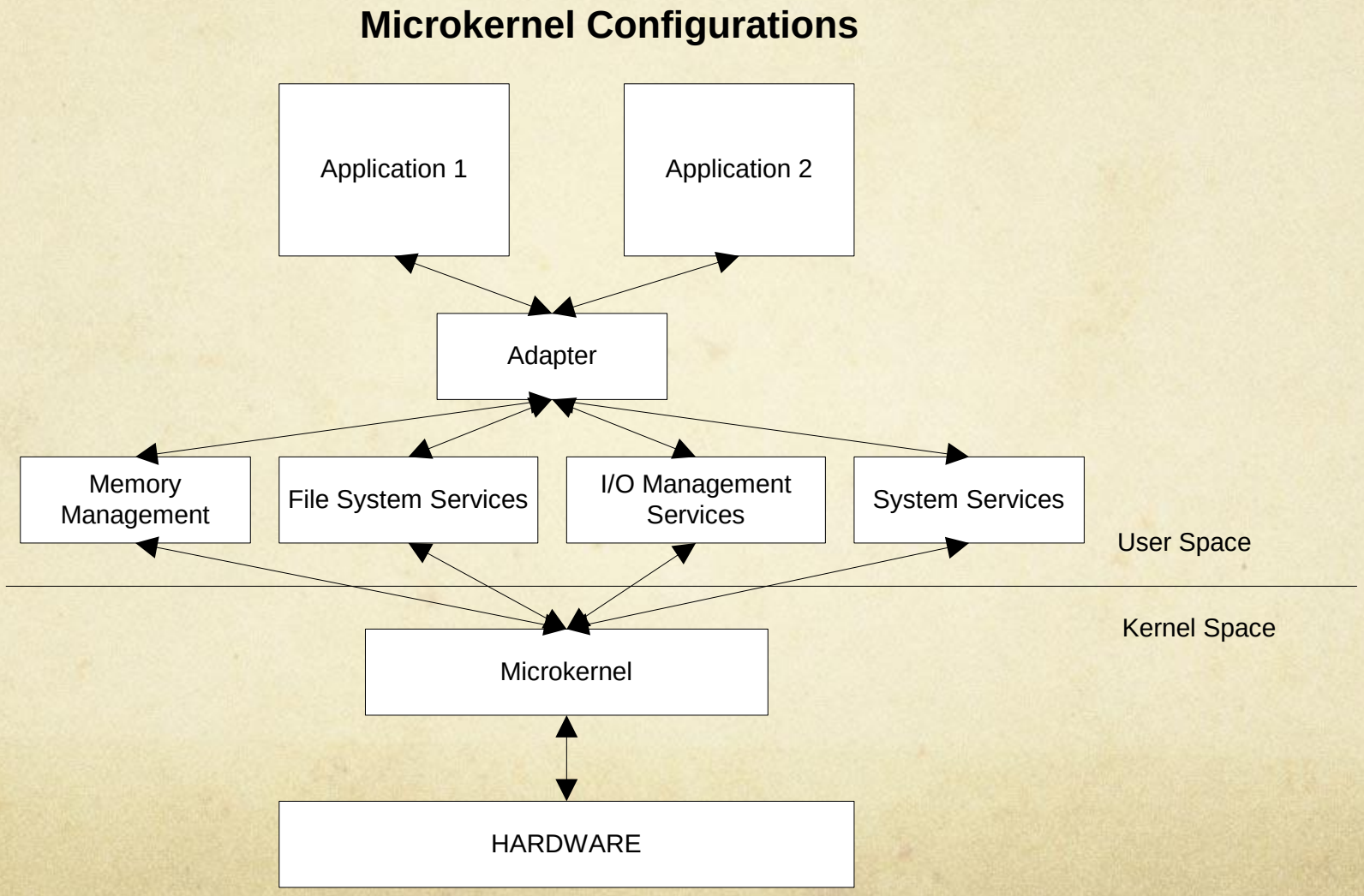
- The strategy is to define a common core services layer, which provides fundamental interfaces to the hardware layer
- Any additional services provided to applications is implemented as add-on modules to this fundamental core, via defined interfaces
- Modules can be added, removed, or replaced without affecting the core functionality of the microkernel
- Benefits are created because the core is minimalist and efficient, and yet expandable

Traditional Monolithic Kernels

Traditional Kernel Configurations



Microkernel Configurations



Existing Microkernels

- Amoeba (Andrew Tanenbaum) (30 system calls in kernel)
- Mach (Carnegie-Mellon University & DARPA) (153 system calls in kernel)
  - Mach serves as the basis for MAC OS/X and GNU HURD
- Chorus (112 system calls in kernel)
- NT (HAL Layer only) (Executive layer and above is quite large—Windows 2000 has 29 million lines of source code)

Amoeba History

- Amoeba began as a research project by Andrew Tanenbaum in the Netherlands in 1981 as a brand new distributed OS
- The distribution was handled by RPC
- The distributed characteristics were:
  - a user logs into the "system", not a particular node
  - the system is made up of multiple servers containing multiple processors in a "processor pool"
  - user commands are run "on the system", which means the next user command rarely runs on the same processor as the previous one (cf. *amake*)
  - a microkernel runs on each processor in the pool (SMP)

The Amoeba Microkernel

- The Microkernel has four primary functions:
  - manage multiple processes and threads
  - provide low-level memory management support (dynamic segmentation)
  - support interprocess communication
    - point-point (blocking, via Amoeba RPC)
    - group (broadcast via Amoeba RPC)
  - handle low-level I/O (device drivers are *inside* the kernel, known as *internal servers*)

Microkernel External Servers

- Other "traditional" OS services (filesystems, network support (TCP/IP or FLIP), etc.) are run as servers on top of the microkernel
- This provides one key benefit:
  - multiple *renditions* of servers can run simultaneously for different user populations, delivering:
    - flexibility (freedom of choice)
    - redundancy

Amoeba Server Objects

- Servers host Amoeba *objects*, which are protected encapsulations of data & function which perform some functionality on that data
- Objects are passive entities (they are operated on by servers)
- Objects are managed by server processes, and communicated with by clients over RPC
- The microkernel provides location services for clients looking for objects
- Amoeba objects are garbage collected

Amoeba Objects

- Examples of Amoeba objects include:
  - filesystem objects (think Unix inode)
  - process objects (think Unix process)
  - thread objects (think POSIX thread)
  - memory segment objects

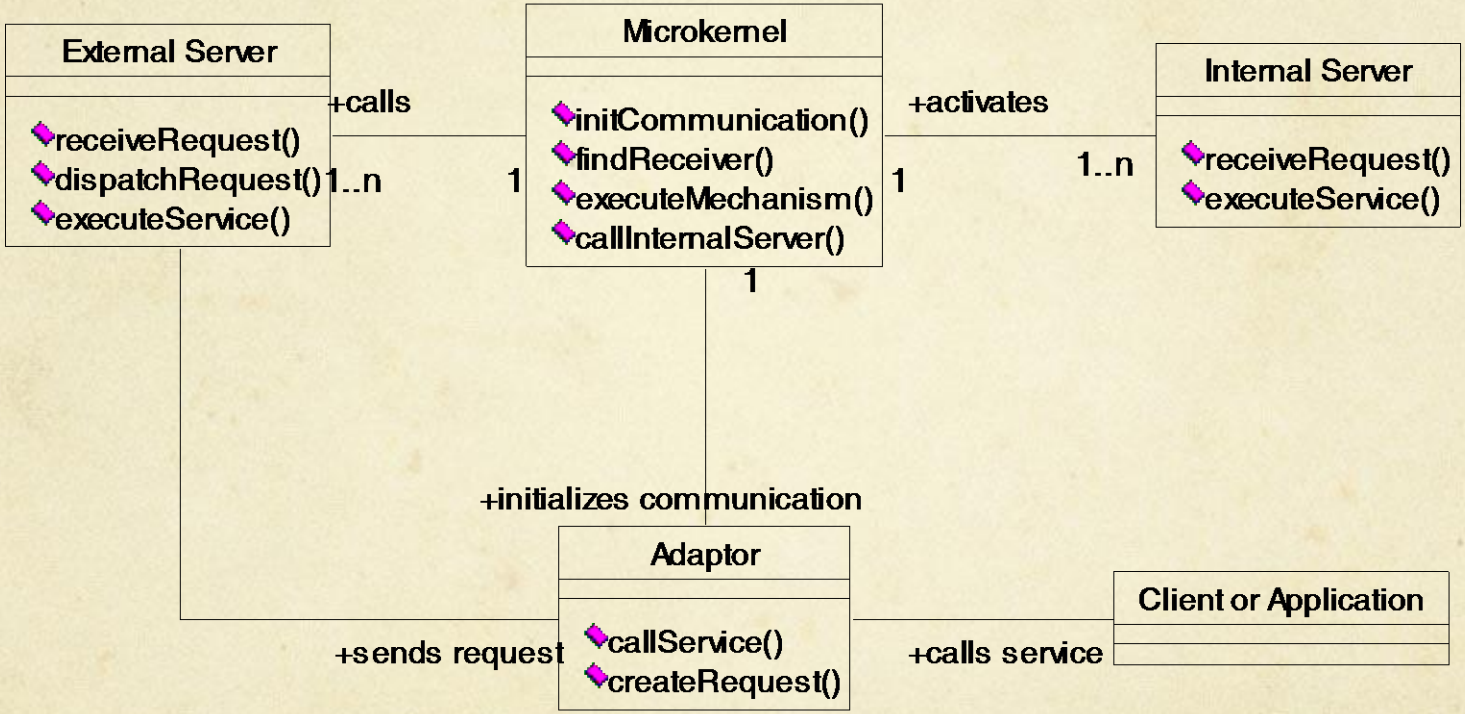
Distributed Client-Server Model

- Amoeba is based on the client-server model
- Servers have stubs (library-based) for clients
- Clients call stub functions which marshal the parameters, send the message and block until the implementation of the server has processed the message and returned
- A stub compiler is available

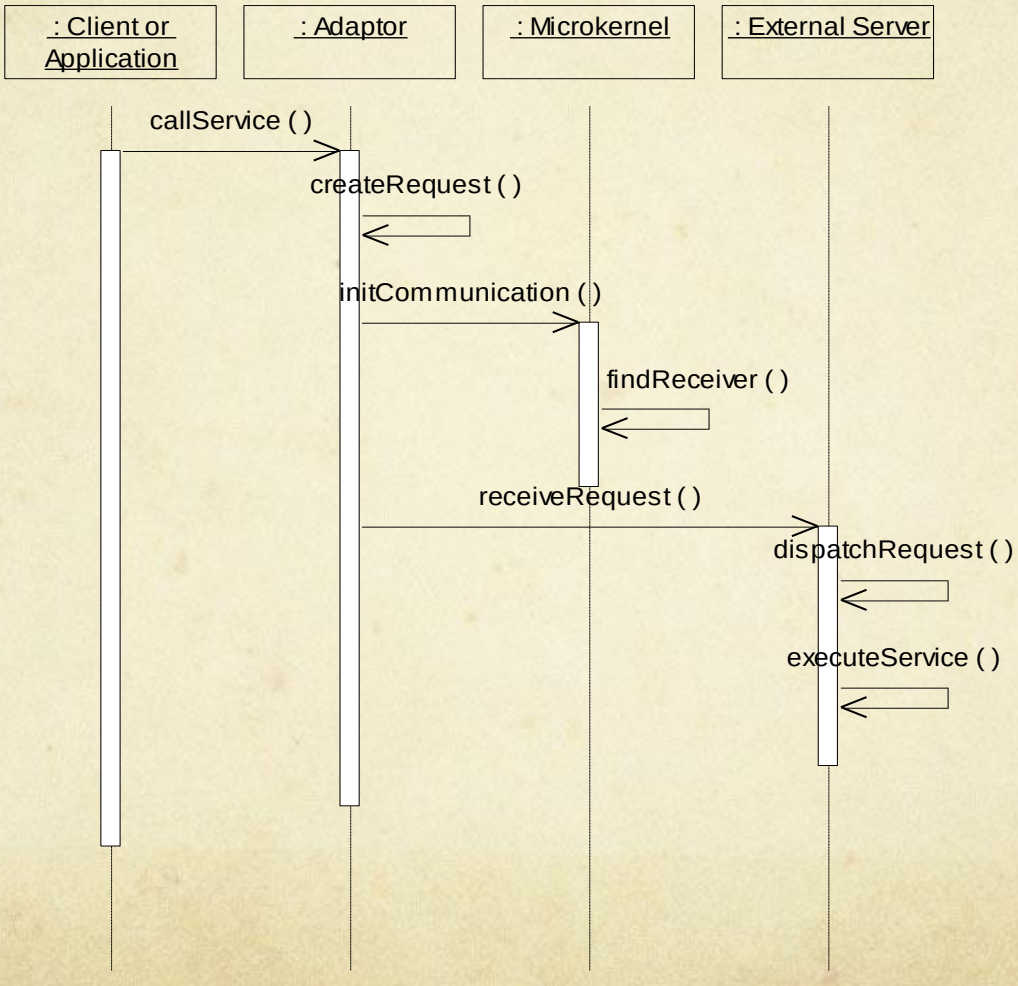
Amoeba Files

- A file is an object that has a capability as well as a function/data association
- The filesystem is a collection of server processes, including the “bullet” server (fast), and the directory server
- A client creates a file, passes it to the bullet server over RPC, and the server stores it “somewhere” and returns a capability (ticket) for later access
- The directory server maps names to capabilities
- Servers hosting capabilities are located by a *broadcast* message

Microkernel UML



Client Call


: Client or Application
: Adaptor
: Microkernel
: External Server
callService ( )
createRequest ( )
initCommunication ( )
findReceiver ( )
receiveRequest ( )
dispatchRequest ( )
executeService ( )

Placeholder for Mach Slides