CISC 322

Software Architecture

Lecture 03:

Architecture Styles

Ahmed E. Hassan

Topics in Architectural Design

Material drawn from [Bass et al. 98, Shaw96, CORBA98, CORBA96, IBM98, Gamma95, JavaIDL98]

[Slides by Spiros Mancoridis]

Software Architecture Topics

- Terminology and Motivation
- Abstraction
- Intuition About Architecture:
  - Hardware
  - Network
  - Building Architecture

Software Architecture Topics

- Architectural Styles of Software Systems:
  – Repository
  – Pipe and Filter
    - Case Study of Compiler Architecture
  – Object-Oriented
  – Implicit Invocation
  – Layered
  – Interpreter
  – Process-Control
  – Client/Server

Software Architecture Topics

- Technologies for Distributed Architectures:
  - IBM’s MQSeries
  - OMG’s CORBA

What is Software Architecture?

- The software architecture of a program or computing system is the structure (or structures) of the system.
- The structures comprise:
  - Software *components*
  - *Externally visible properties* of the components
  - *Relationships* between the components

Externally Visible Properties of Components

- Externally visible properties refers to those assumptions other components can make of a component, such as:
  - Provided services
  - Performance characteristics
  - Fault handling
  - Shared resource usage
  - Et cetera

A Software Architecture is an Abstraction

- An architecture is an abstraction of a system that suppresses details of components that do not affect how they:
  - Use
  - Are used by
  - Relate to
  - Interact with

other components.

Can a system have more that one structure?

- Yes, no one structure holds the claim to being *the* architecture.
- Below are some examples of structures:
  - Module decomposition
  - Inheritance hierarchy
  - Call graph
  - Objects and message passing at runtime
  - Build dependencies
  - Et cetera

Does Every System have an Architecture?

- *Yes.*
- For small systems the architecture may be trivial.
- For large systems it definitely exists in the software product, but may not have been documented.

Are box-and-line diagrams descriptions of Software Architecture?

- No.
- A description of the behavior of each component is part of the architecture.
- In box-and-line diagrams, readers imagine the behavior of each component by interpreting the labels of the boxes & lines.
- One must document the extent that a component's behavior influences how another component must be written to interact with it.

Why is Software Architecture Important?

- Communication among stakeholders:
  - Customers, managers, designers, programmers.
- Documentation of early design decisions:
  - Constraints on implementation
  - Organizational structure
  - Guides evolutionary prototyping
- Transferable abstraction of a system to similar systems (reuse):
  - Program families share a common architecture
  - Architecture can be the basis for training.

Architectural Structures (Module Structure)

- **Components**: *work assignments*.
  - Work assignments have products associated with them:
    - Interface specifications
    - Code
    - Test plans, etc.
- **Relations**: *is-a-submodule-of*
- **Use**: Allocating a project's labor and other resources during development and maintenance.

Architectural Structures (Conceptual or Logical Structure)

- **Components**: Abstractions of the system's *functional requirements*.
- **Relations**: *shares-data-with*
- **Use**: Understanding the interactions between units in the problem space.

Architectural Structures (Process or Coordination Structure)

- **Components**: Processes or threads.
- **Relations**:
  - *Synchronizes-with*
  - *Can't-run-without*
  - *Can't-run-with*
  - *Preempts*, et cetera
- **Use**: Modeling dynamic aspects of a running system.

Architectural Structures (Physical Structure)

- **Components**: *Hardware* (computers, networks, etc.)
- **Relations**: *communicates-with*
- **Use**: Create models to reason about performance, availability, security, etc.

Architectural Structures (Uses Structure)

- **Components**: *procedures* or *modules*
- **Relations**: *assumes-the-correct-presence-of*
- **Use**: To model system extendibility and incremental system building (e.g., Makefile dependencies).

Architectural Structures (Calls Structure)

- **Components**: *Procedures*
- **Relations**: *calls*
- **Uses**: To model trace of execution in a program.

Architectural Structures (Data Flow Structure)

- **Components**: *Programs* or *modules*
- **Relations**: *transmits-data-to*
- **Use**: To model data transmission which can aid requirements traceability.

Architectural Structures (Class Structure)

- **Components**: *classes* and *interfaces*
- **Relations**: *inherits-from, implements*
- **Use**: To model collections of similar behavior and parameterizes differences.

The Importance of Structures

- Structures are important because they "boil away" details about the software that are independent of the concern reflected by the abstraction.
- Each structure provides a useful perspective of the system.
- Sometimes the term *view* is used instead of *structure*.

Abstraction

Abstraction

- One characterization of progress in software development has been the regular increase in levels of abstraction:
  - *I.e.*, the size of a software designer's building blocks.

Abstraction (Cont'd)

- **<u>Early 1950s:</u>** Software was written in machine language:
  - programmers placed instructions and data individually and explicitly in the computer's memory
  - insertion of a new instruction in a program might require hand checking the entire program to update references to data and instructions

Assemblers

- Some machine code programming problems were solved by adding a level of abstraction between the program and the machine:
  - **Symbolic Assemblers:**
    - Names used for operation codes and memory addresses.
    - Memory layout and update of references are automated.
  - **Macro Processors:**
    - Allow a single symbol to stand for a commonly used sequence of instructions.

Programming Languages

- **Late 1950s:** The emerging of the first high-level programming languages. Well understood patterns are created from notations that are more like mathematics than machine code.
  - evaluation of arithmetic expressions
  - procedure invocation
  - loops and conditionals

Programming Languages (Cont'd)

- **FORTRAN** becomes the first widely used programming language.
- **Algol** and its successors followed with higher-levels of abstraction for representing data (types).

Abstract Data Types

- **<u>Late 1960s and 1970s:</u>** Programmers shared an intuition that good data structure design will ease the development of a program.
- This intuition was converted into theories of modularization and information hiding.
  - Data and related code are encapsulated into modules.
  - Interfaces to modules are made explicit.

Abstract Data Types (Cont'd)

- **Programming Languages:**
  - Modula
  - Ada
  - Euclid
- **Module Interconnection Languages:**
  - MIL75
  - Intercol

Software Architecture

- As the size and complexity of software systems increases, the design problem goes beyond algorithms and data structures.
- Designing and specifying the overall system structure (**Software Architecture**) emerges as a new kind of problem.

Software Architecture Issues

- Organization and global control structure.
- Protocols of communication, synchronization, and data access.
- Assignment of functionality to design elements.
- Physical distribution of data and processes.
- Selection among design alternatives.

State of Practice

- There is not currently a well-defined terminology or notation to characterize architectural structures.
- However, good software engineers make common use of architectural principles when designing software.
- These principles represent rules of thumb or patterns that have emerged informally over time. Others are more carefully documented as industry standards.

Descriptions of Architectures

- *“Camelot is based on the* ***client-server model*** *and uses remote procedure calls both locally and remotely to provide communication among applications and servers.”*

Descriptions of Architectures (Cont'd)

- *"**Abstraction layering** and system decomposition provide the appearance of system uniformity to clients, yet allow Helix to accommodate a diversity of autonomous devices. The architecture encourages a **client-server model** for the structuring of applications."*

Descriptions of Architectures (Cont'd)

- *"We have chosen a* ***distributed***, ***object-oriented*** *approach to managing information."*

Descriptions of Architectures (Cont'd)

- *"The easiest way to make a canonical sequential compiler into a concurrent compiler is to **pipeline** the execution of the compiler phases over a number of processors. A more effective way is to split the source code into many segments, which are concurrently processed through the various phases of compilation (by multiple compiler processes) before a final, merging pass recombines the object code into a single program."*

Some Standard Architectures

- **ISO/OSI Reference Model** is a layered network architecture.
- **X Window System** is a distributed windowed user interface architecture based on event triggering and callbacks.
- **NIST/ECMA Reference Model** is a generic software engineering environment architecture based on layered communication substrates.

Intuition About Architecture

Intuition About Architecture

- It is interesting that we have so few *named* software architectures. This is not because there are so few architectures, but so many.
- Next we look at several architectural disciplines in order to develop an intuition about software architecture:
  - Hardware Architecture
  - Network Architecture
  - Building Architecture

Hardware Architecture

- **RISC** machines emphasize the instruction set as an important feature.
- **Pipelined** and **multi-processor** machines emphasize the configuration of architectural pieces of the hardware.

Differences and Similarities Between SW & HW Architectures

- **<u>Differences:</u>**
  - Relatively (to software) small number of design elements.
  - Scale is achieved by replication of design elements.
- **<u>Similarities:</u>**
  - We often configure software architectures in ways analogous to hardware architectures. (*e.g.*, we create multi-process software and use pipelined processing).

Network Architecture

- Networked architectures abstract the design elements of a network into nodes and connections.
- Topology is the most emphasized aspect:
  - Star networks
  - Ring networks
  - Manhattan Street networks
- Unlike software architectures, in network architectures only few topologies are of interest.

Building Architecture

- **Multiple Views:** skeleton frames, detailed views of electrical wiring, etc.
- **Architectural Styles:** Classical, Romanesque, Colonial, and so on.
- **Materials:** One does not build a skyscraper using wooden posts and beams.

Architectural Styles of Software Systems

Architectural Styles of Software Systems

- An **Architectural Style** defines a family of systems in terms of a pattern of structural organization. It determines:
  - the **vocabulary** of components and connectors that can be used in instances of that style
  - a set of **constraints** on how they can be combined. For example, one might constrain:
    - the topology of the descriptions (*e.g.*, no cycles).
    - execution semantics (*e.g.*, processes execute in parallel).

Determining an Architectural Style

- We can understand what a style is by answering the following questions:
  - What is the **structural pattern**? (*i.e.*, components, connectors, constraints)
  - What is the underlying **computational model**?
  - What are the essential **invariants** of the style?
  - What are some common **examples** of its use?
  - What are the **advantages** and **disadvantages** of using that style?
  - What are some of the common **specializations** of that style?

Describing an Architectural Style

- The architecture of a specific system is a collection of:
  - computational components
  - description of the interactions between these components (connectors)

Describing an Architectural Style (Cont'd)

- Software architectures are represented as graphs where **nodes** represent components:
  - procedures
  - modules
  - processes
  - tools
  - databases
- and **edges** represent connectors:
  - procedure calls
  - event broadcasts
  - database queries
  - pipes

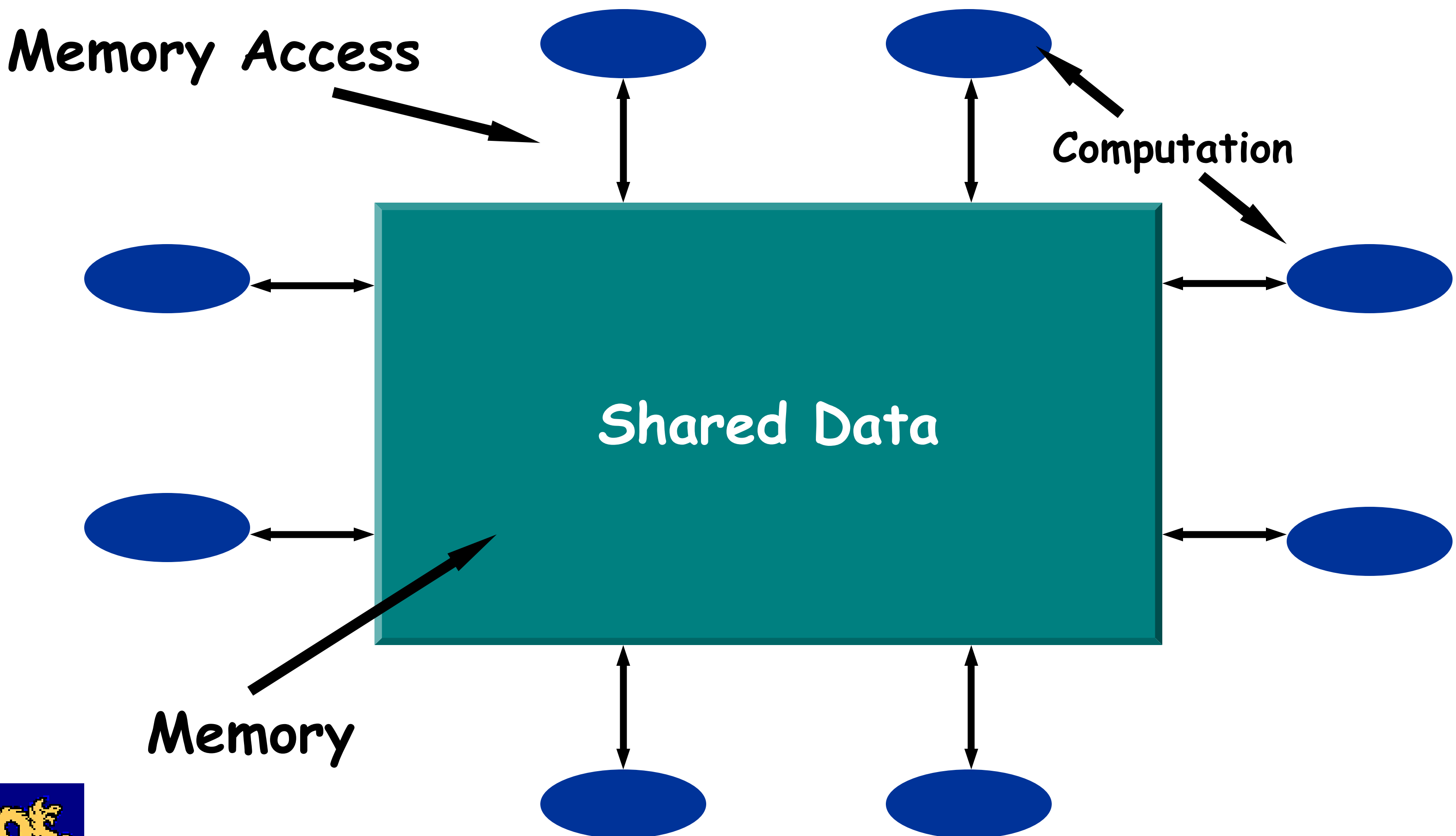
Repository Style

- Suitable for applications in which the central issue is establishing, augmenting, and maintaining a complex central body of information.
- Typically the information must be manipulated in a variety of ways. Often long-term persistence is required.

Repository Style (Cont'd)

- **Components:**
  - A central data structure representing the current state of the system.
  - A collection of independent components that operate on the central data structure.
- **Connectors:**
  - Typically procedure calls or direct memory accesses.

Repository Style (Cont'd)



Repository Style Specializations

- Changes to the data structure trigger computations.
- Data structure in memory (persistent option).
- Data structure on disk.
- Concurrent computations and data accesses.

Repository Style Examples

- Information Systems
- Programming Environments
- Graphical Editors
- AI Knowledge Bases
- Reverse Engineering Systems

Repository Style Advantages

- **Efficient** way to store large amounts of data.
- **Sharing** model is published as the repository schema.
- Centralized **management**:
  - backup
  - security
  - concurrency control

Repository Style Disadvantages

- Must **agree on a data model** a priori.
- Difficult to **distribute data**.
- Data **evolution is expensive**.

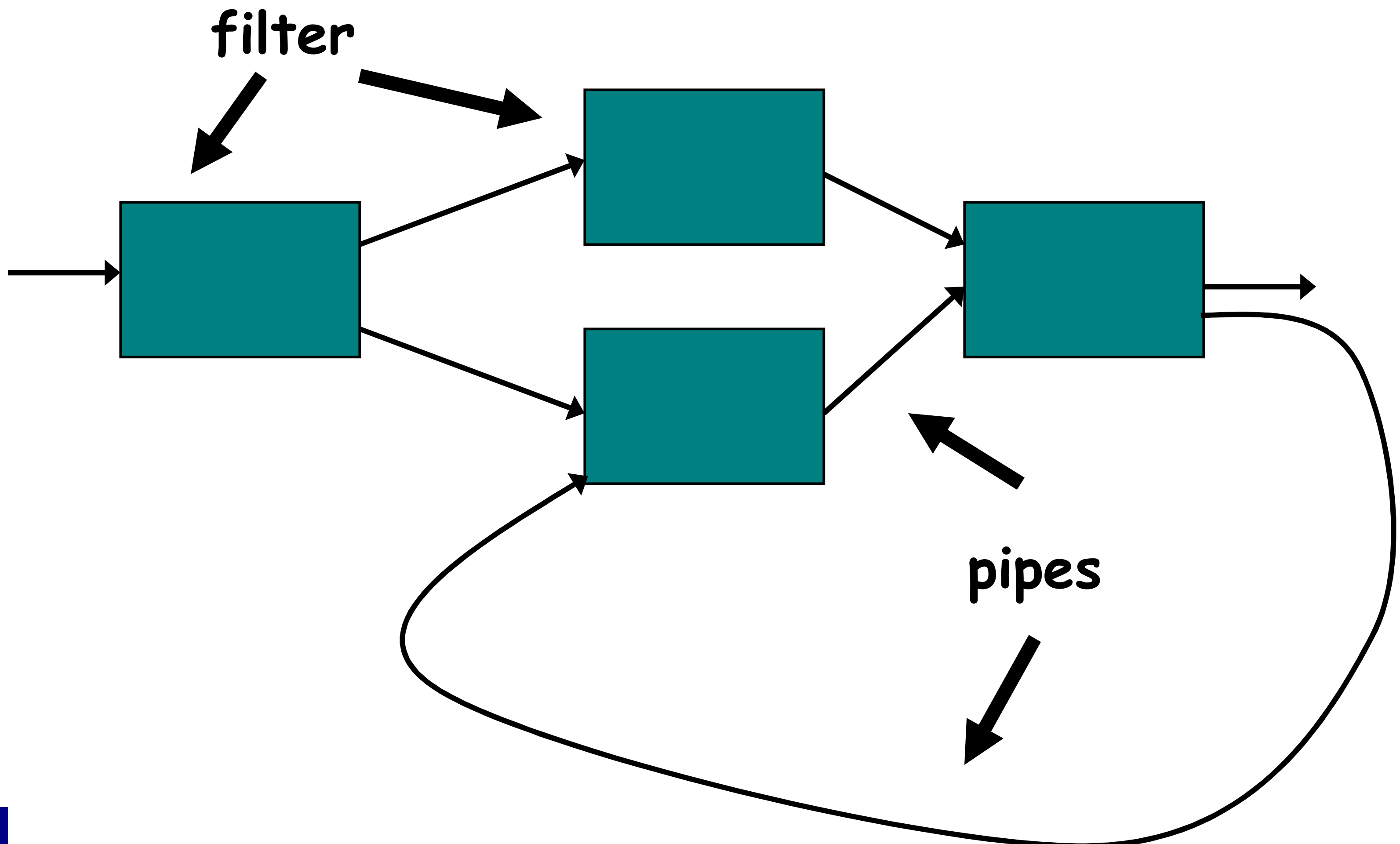
Pipe and Filter Architectural Style

- Suitable for applications that require a defined series of independent computations to be performed on data.
- A component reads streams of data as input and produces streams of data as output.

Pipe and Filter Architectural Style (Cont'd)

- **Components:** called **filters**, apply local transformations to their input streams and often do their computing incrementally so that output begins before all input is consumed.
- **Connectors:** called **pipes**, serve as conduits for the streams, transmitting outputs of one filter to inputs of another filter.

Pipe and Filter Architectural Style (Cont'd)



Pipe and Filter Invariants

- Filters **do not share state** with other filters.
- Filters **do not know the identity** of their upstream or downstream filters.

Pipe and Filter Specializations

- **<u>Pipelines:</u>** Restricts topologies to linear sequences of filters.
- **<u>Batch Sequential:</u>** A degenerate case of a pipeline architecture where each filter processes all of its input data before producing any output.

Pipe and Filter Examples

- **Unix Shell Scripts:** Provides a notation for connecting Unix processes via pipes.
  - *cat file | grep Erroll | wc -l*
- **Traditional Compilers:** Compilation phases are pipelined, though the phases are not always incremental. The phases in the pipeline include:
  - *lexical analysis* + *parsing* + *semantic analysis* + *code generation*

Pipe and Filter Advantages

- **Easy to understand** the overall input/output behavior of a system as a simple composition of the behaviors of the individual filters.
- They **support reuse**, since any two filters can be hooked together, provided they agree on the data that is being transmitted between them.

Pipe and Filter Advantages (Cont'd)

- Systems can be **easily maintained and enhanced**, since new filters can be added to existing systems and old filters can be replaced by improved ones.
- They permit certain kinds of **specialized analysis**, such as throughput and deadlock analysis.
- The naturally **support concurrent execution**.

Pipe and Filter Disadvantages

- Not good choice for **interactive systems**, because of their transformational character.
- Excessive parsing and unparsing leads to **loss of performance** and **increased complexity** in writing the filters themselves.

Case Study: Architecture of a Compiler

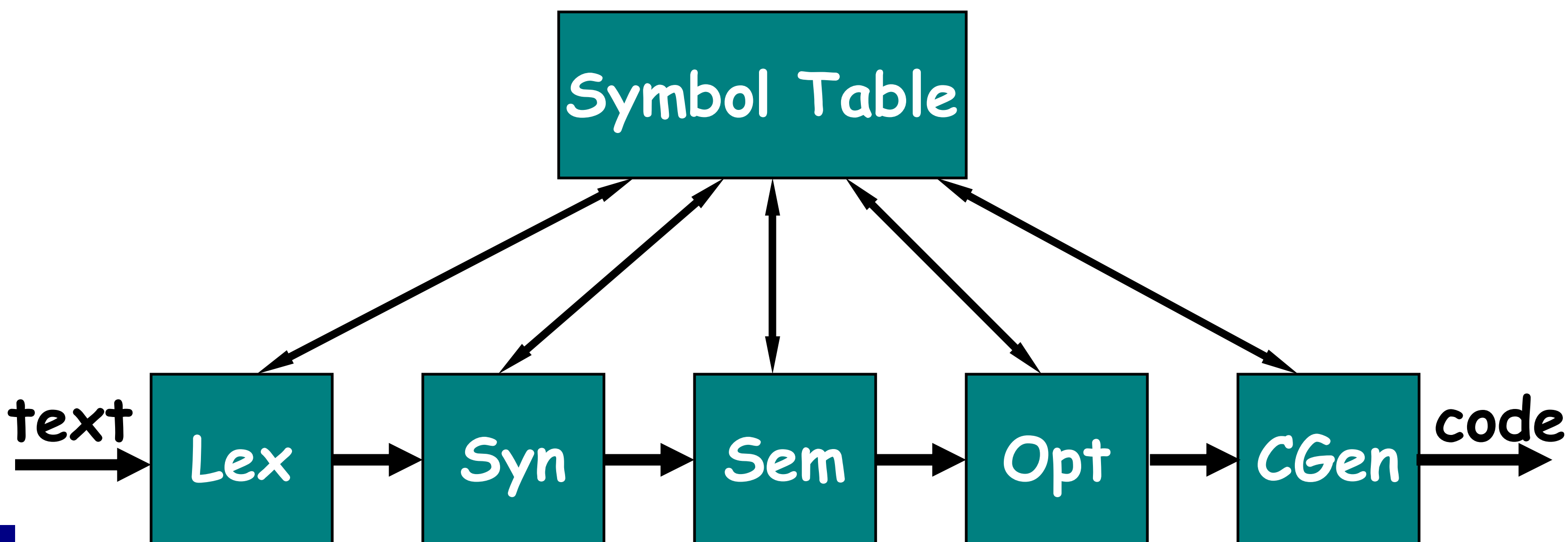
- The architecture of a system can change in response to improvements in technology.
- This can be seen in the way we think about compilers.

Early Compiler Architectures

- In the 1970s, compilation was regarded as a sequential (batch sequential or pipeline) process:

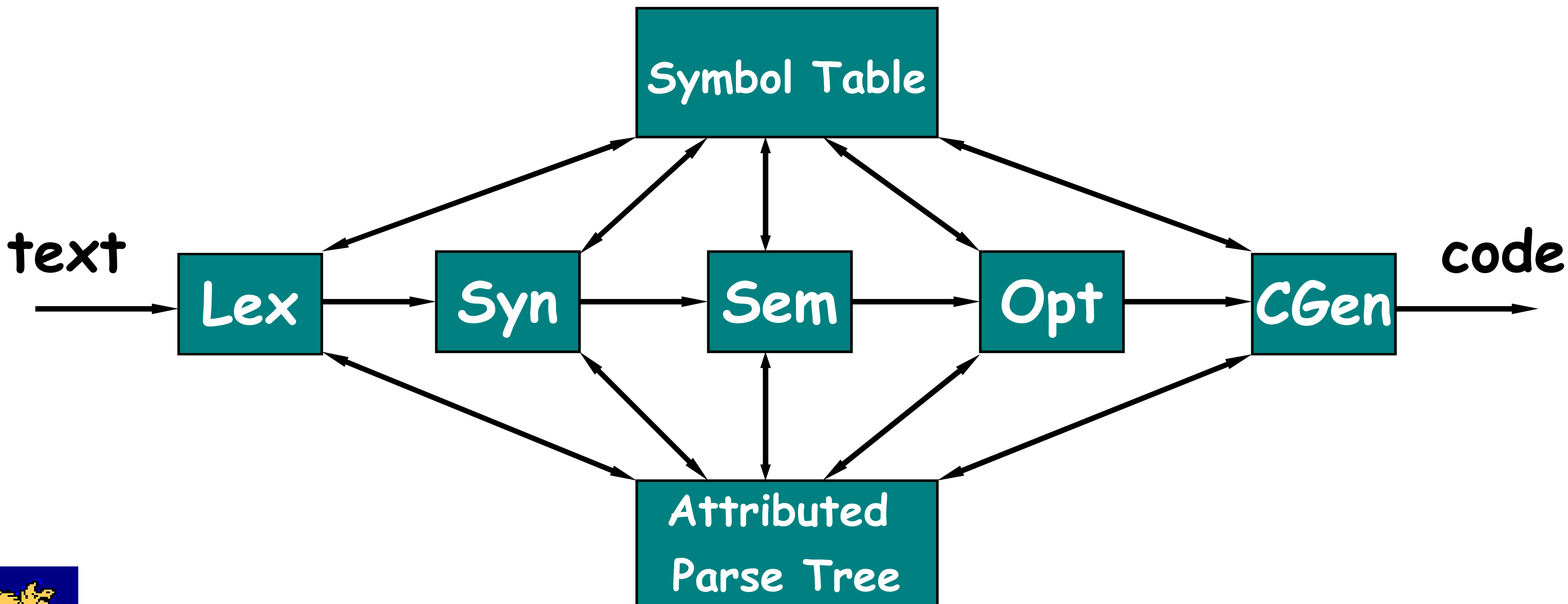
Early Compiler Architectures

- Most compilers create a separate symbol table during lexical analysis and used or updated it during subsequent passes.



Modern Compiler Architectures

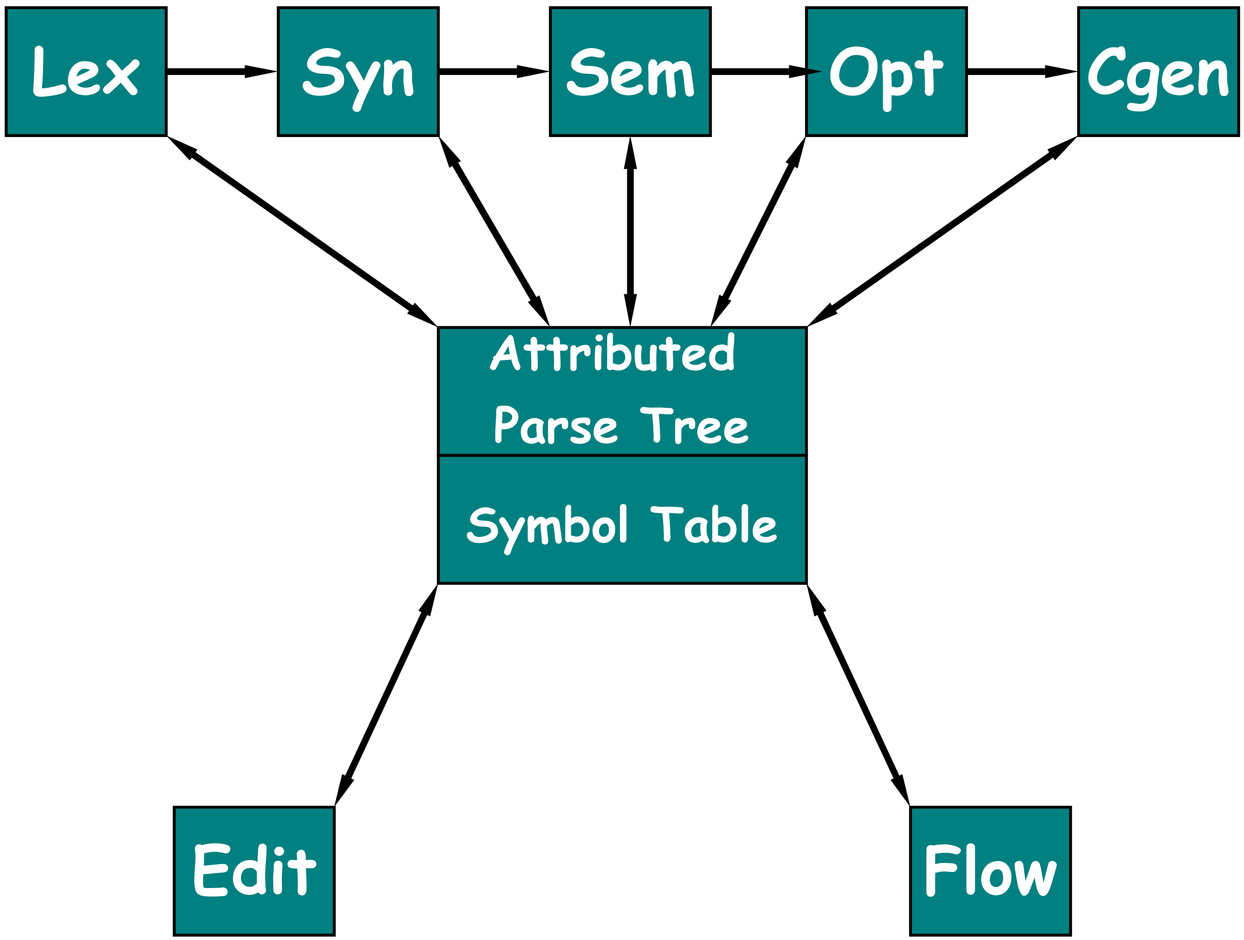
- Later, in the mid 1980s, increasing attention turned to the intermediate representation of the program during compilation.



Hybrid Compiler Architectures

- The new view accommodates various tools (*e.g.*, syntax-directed editors) that operate on the internal representation rather than the textual form of a program.
- Architectural shift to a **repository** style, with elements of the **pipeline** style, since the order of execution of the processes is still predetermined.

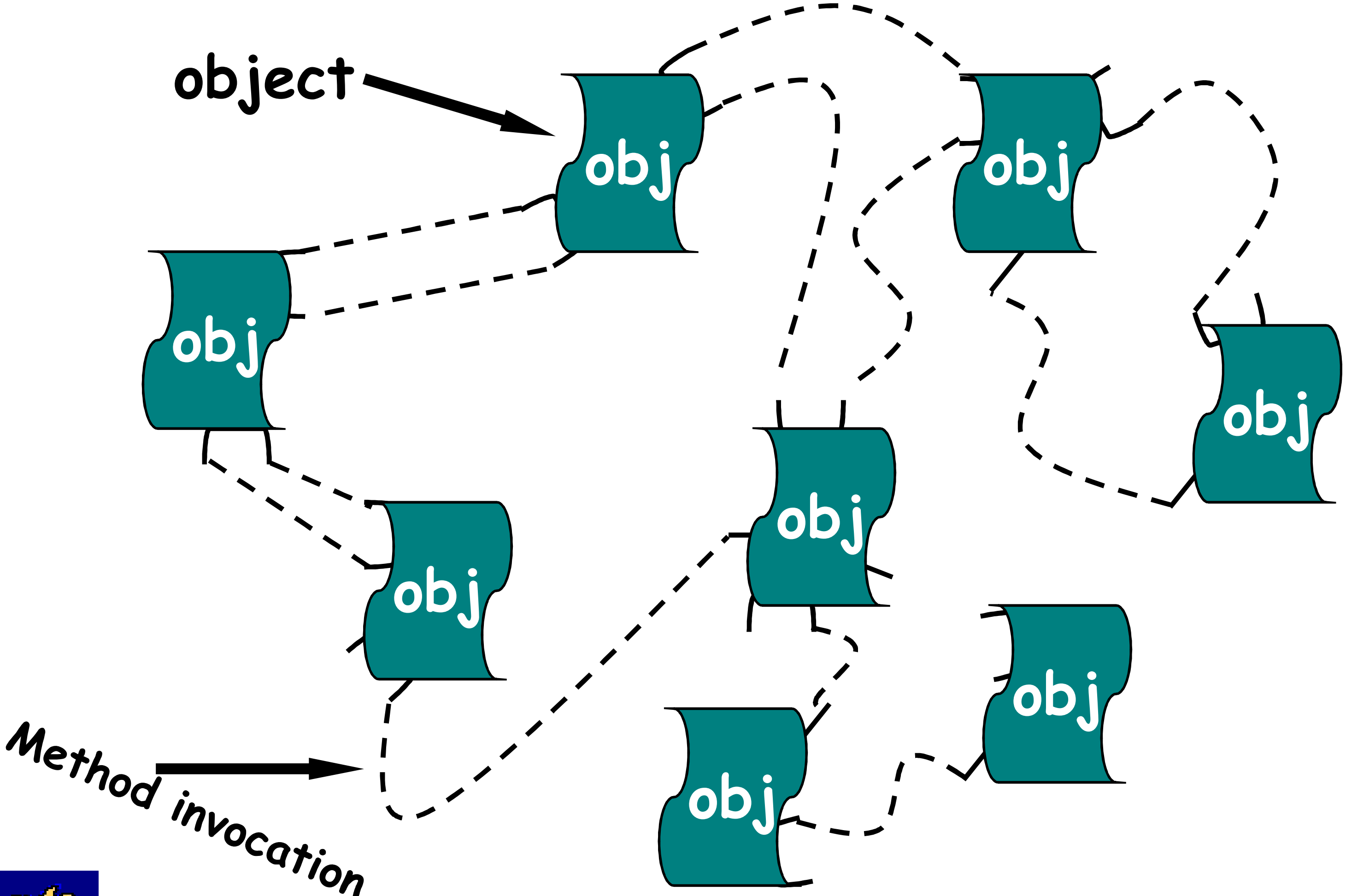
Hybrid Compiler Architectures



Object-Oriented Style

- Suitable for applications in which a central issue is identifying and protecting related bodies of information (data).
- Data representations and their associated operations are encapsulated in an **abstract data type**.
- **Components:** are **objects**.
- **Connectors:** are function and procedure invocations (**methods**).

Object-Oriented Style (Cont'd)



Object-Oriented Invariants

- Objects are responsible for preserving the **integrity** (*e.g.*, some invariant) of the data representation.
- The data representation is **hidden** from other objects.

Object-Oriented Specializations

- Distributed Objects
- Objects with Multiple Interfaces

Object-Oriented Advantages

- Because an object hides its data representation from its clients, it is possible to **change the implementation without affecting those clients**.
- Can **design** systems as collections of autonomous interacting agents.

Object-Oriented Disadvantages

- In order for one object to interact with another object (via a method invocation) the first object must know the **identity** of the second object.
  – Contrast with *Pipe and Filter* Style.
  – When the identity of an object changes, it is necessary to modify all objects that invoke it.
- Objects cause **side effect problems**:
  – *E.g.*, *A* and *B* both use object *C*, then *B*'s effects on *C* look like unexpected side effects to *A*.

Implicit Invocation Style

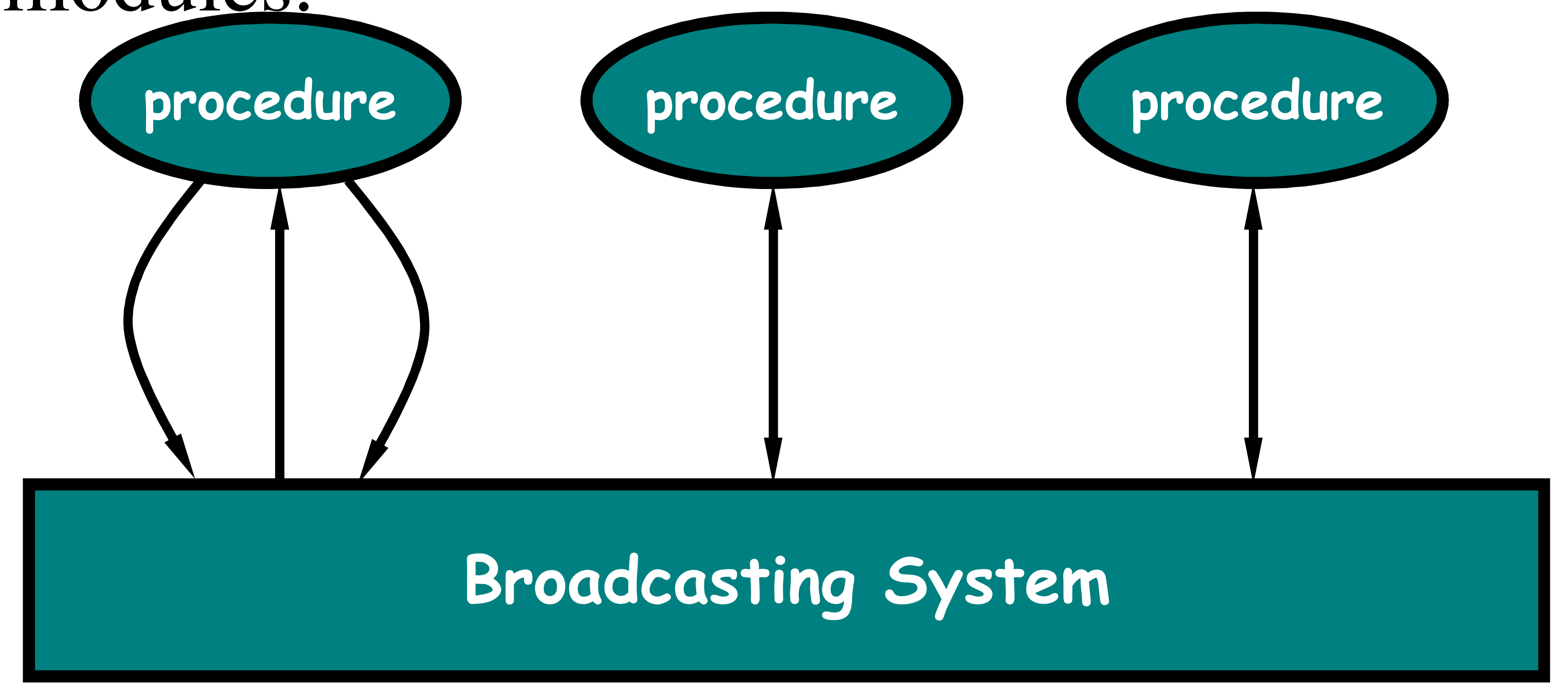
- Suitable for applications that involve loosely-coupled collection of components, each of which carries out some operation and may in the process enable other operations.
- Particularly useful for applications that must be reconfigured "on the fly":
  - Changing a service provider.
  - Enabling or disabling capabilities.

Implicit Invocation Style (Cont'd)

- Instead of invoking a procedure directly ...
  - A **component** can announce (or broadcast) one or more events.
  - Other **components** in the system can register an interest in an event by associating a procedure with the event.
  - When an event is announced, the broadcasting system (**connector**) itself invokes all of the procedures that have been registered for the event.

Implicit Invocation Style (Cont'd)

- An event announcement "implicitly" causes the invocation of procedures in other modules.



Implicit Invocation Invariants

- Announcers of events do not know which components will be affected by those events.
- Components cannot make assumptions about the order of processing.
- Components cannot make assumptions about what processing will occur as a result of their events (perhaps no component will respond).

Implicit Invocation Specializations

- Often connectors in an implicit invocation system include the **traditional procedure call** in addition to the bindings between event announcements and procedure calls.

Implicit Invocation Examples

- Used in **programming environments** to integrate tools:
  - Debugger stops at a breakpoint and makes that announcement.
  - Editor responds to the announcement by scrolling to the appropriate source line of the program and highlighting that line.

Implicit Invocation Examples (Cont'd)

- Used to enforce integrity constraints in **database management systems** (called triggers).
- Used in **user interfaces** to separate the presentation of data from the applications that manage that data.

Implicit Invocation Advantages

- Provides strong support for **reuse** since any component can be introduced into a system simply by registering it for the events of that system.
- **Eases system evolution** since components may be replaced by other components without affecting the interfaces of other components in the system.

Implicit Invocation Disadvantages

- When a component announces an event:
  - it has no idea what other components will respond to it,
  - it cannot rely on the order in which the responses are invoked
  - it cannot know when responses are finished

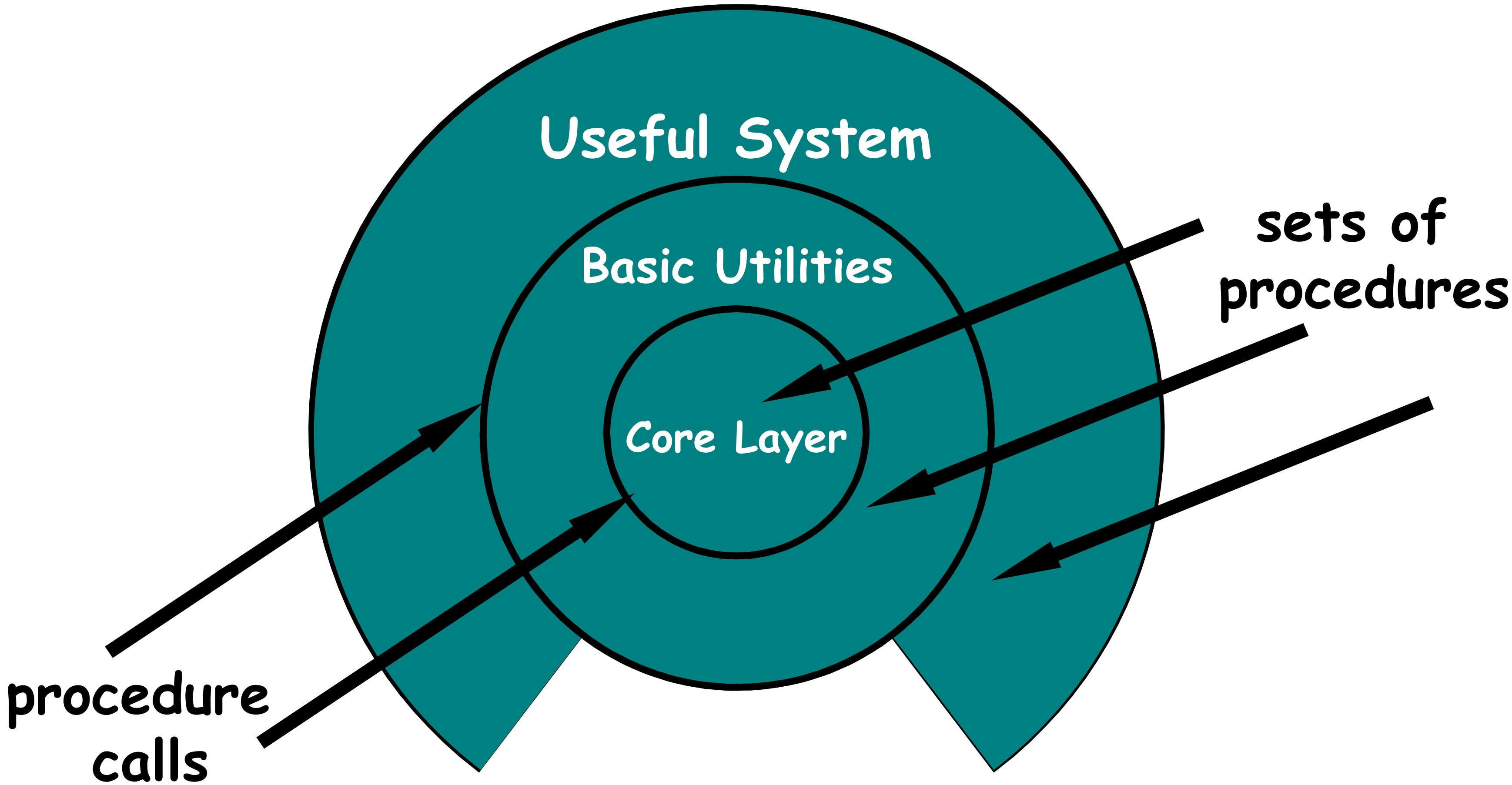
Layered Style

- Suitable for applications that involve distinct classes of services that can be **organized hierarchically**.
- Each layer provides service to the layer above it and serves as a client to the layer below it.
- Only carefully selected procedures from the inner layers are made available (exported) to their adjacent outer layer.

Layered Style (Cont'd)

- **Components:** are typically collections of procedures.
- **Connectors:** are typically procedure calls under restricted visibility.

Layered Style (Cont'd)



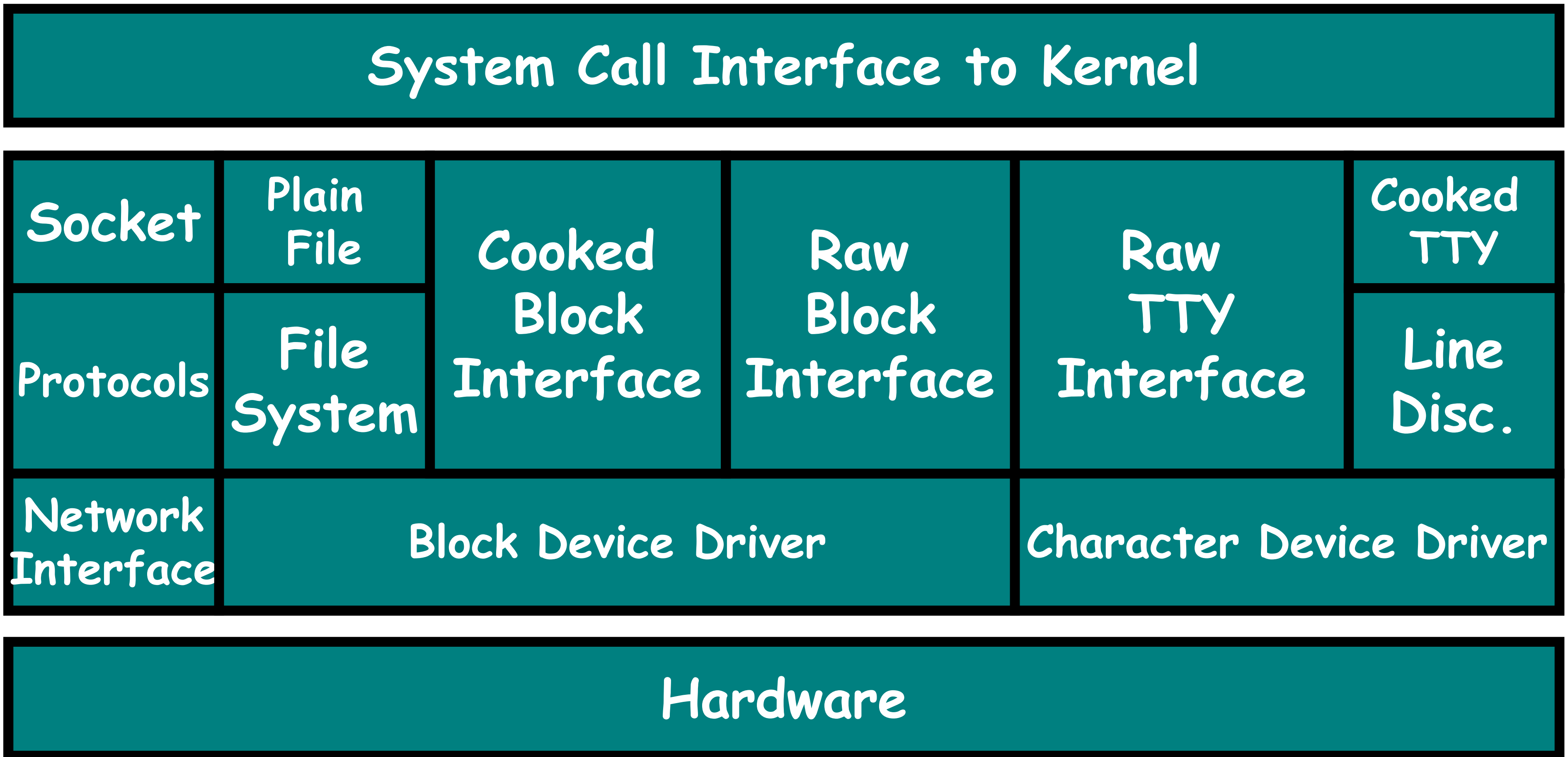
Layered Style Specializations

- Often exceptions are made to **permit non-adjacent layers to communicate directly**.
  - This is usually done for efficiency reasons.

Layered Style Examples

- **Layered Communication Protocols:**
  - Each layer provides a substrate for communication at some level of abstraction.
  - Lower levels define lower levels of interaction, the lowest level being hardware connections (physical layer).
- **Operating Systems**
  - Unix

Unix Layered Architecture



Layered Style Advantages

- **<u>Design:</u>** based on increasing levels of abstraction.
- **<u>Enhancement:</u>** Changes to the function of one layer affects at most two other layers.
- **<u>Reuse:</u>** Different implementations (with identical interfaces) of the same layer can be used interchangeably.

Layered Style Disadvantages

- Not all systems are easily structured in a layered fashion.
- Performance requirements may force the coupling of high-level functions to their lower-level implementations.

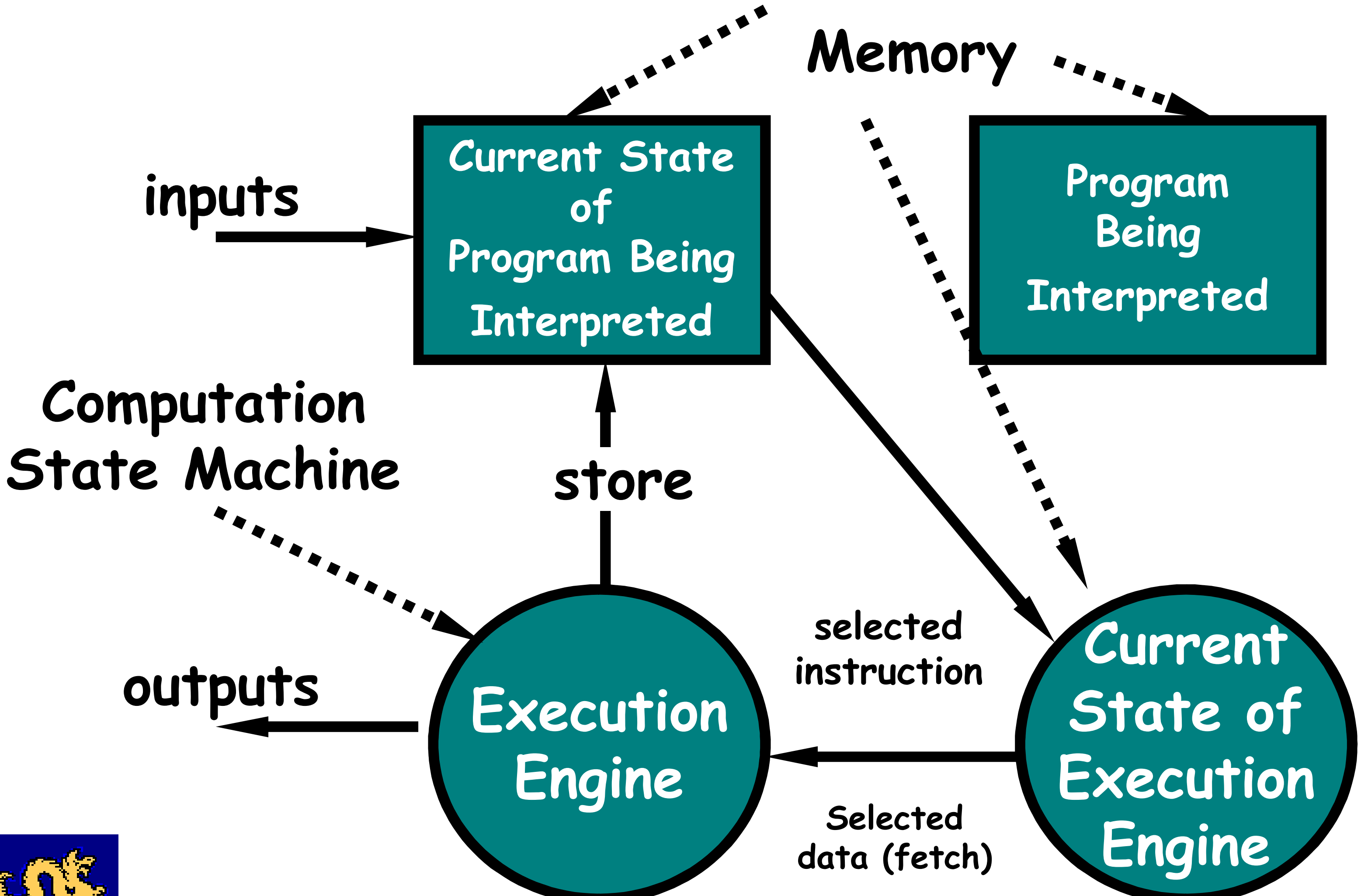
Interpreter Style

- Suitable for applications in which the most appropriate language or machine for executing the solution is not directly available.

Interpreter Style (Cont'd)

- **Components:** include one state machine for the execution engine and three memories:
  - current state of the execution engine
  - program being interpreted
  - current state of the program being interpreted
- **Connectors:**
  - procedure calls
  - direct memory accesses.

Interpreter Style (Cont'd)



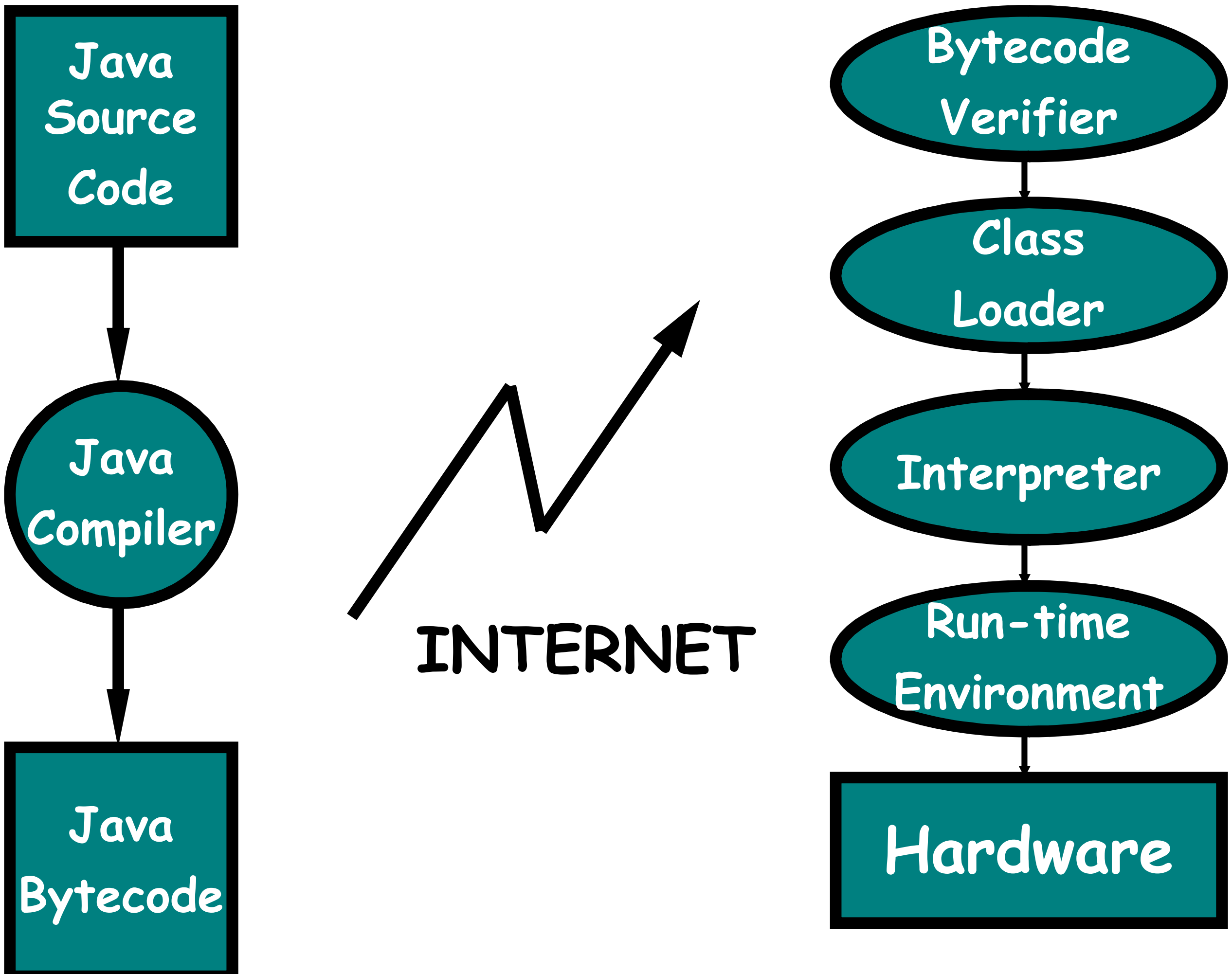
Interpreter Style Examples

- **Programming Language Compilers:**
  - Java
  - Smalltalk
- **Rule Based Systems:**
  - Prolog
  - Coral
- **Scripting Languages:**
  - Awk
  - Perl

Interpreter Style Advantages

- Simulation of non-implemented hardware.
- Facilitates portability of application or languages across a variety of platforms.

Java Architecture



Interpreter Style Disadvantages

- Extra level of indirection **slows** down execution.
- Java has an option to compile code.
  - JIT (Just In Time) compiler.

Process-Control Style

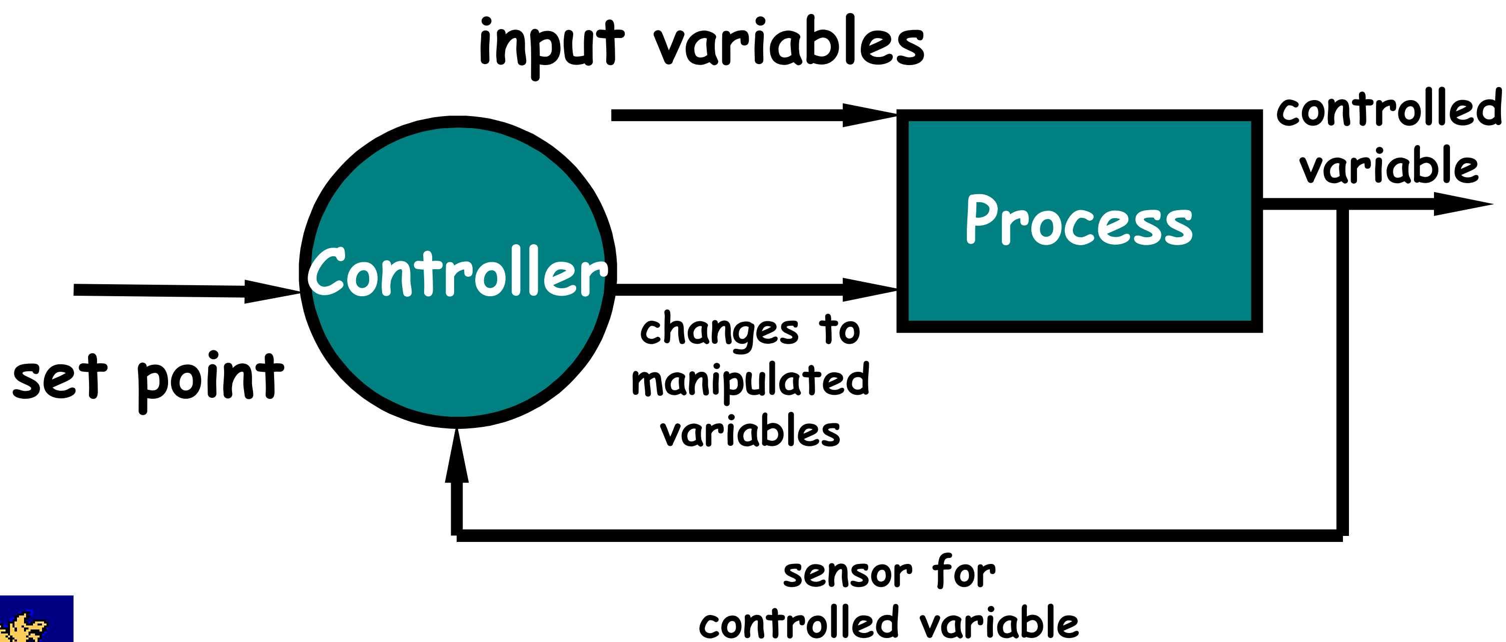
- Suitable for applications whose purpose is to maintain specified properties of the outputs of the process at (sufficiently near) given reference values.
- **Components:**
  - **Process Definition** includes mechanisms for manipulating some process variables.
  - **Control Algorithm** for deciding how to manipulate process variables.

Process-Control Style (Cont'd)

- **<u>Connectors:</u>** are the data flow relations for:
  - **Process Variables:**
    - *Controlled variable* whose value the system is intended to control.
    - *Input variable* that measures an input to the process.
    - *Manipulated variable* whose value can be changed by the controller.
  - **Set Point** is the desired value for a controlled variable.
  - **Sensors** to obtain values of process variables pertinent to control.

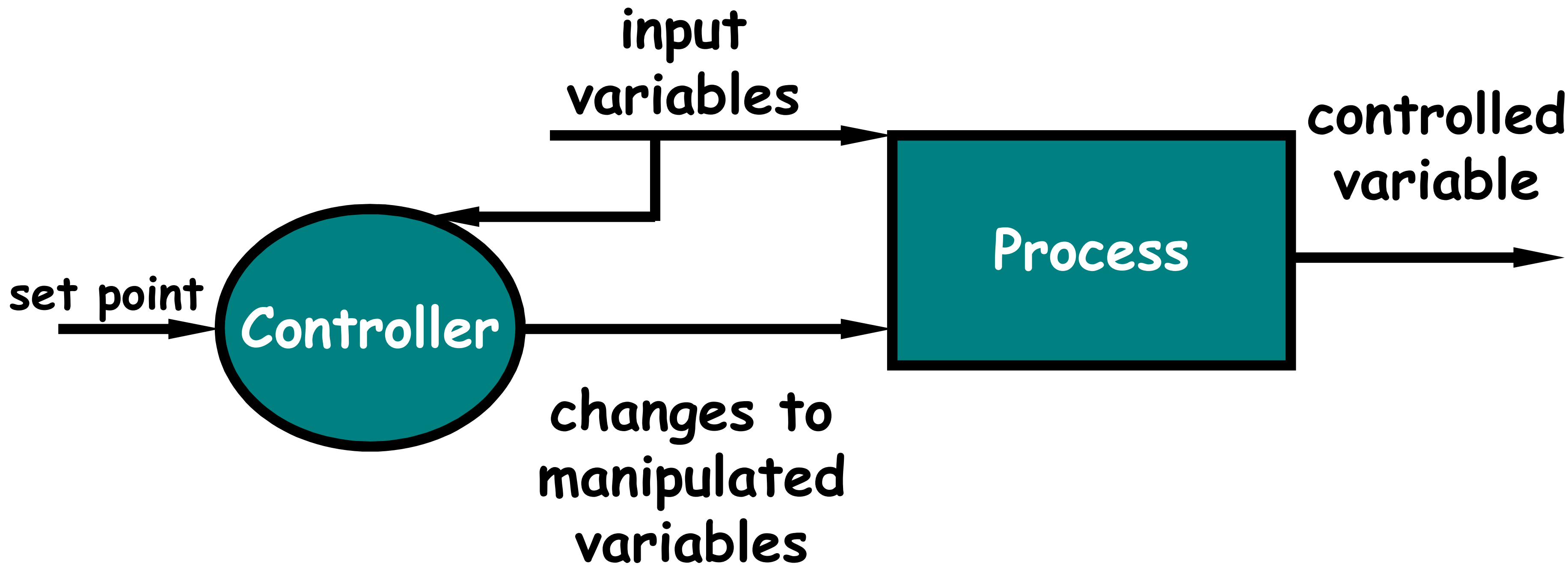
Feed-Back Control System

- The controlled variable is measured and the result is used to manipulate one or more of the process variables.



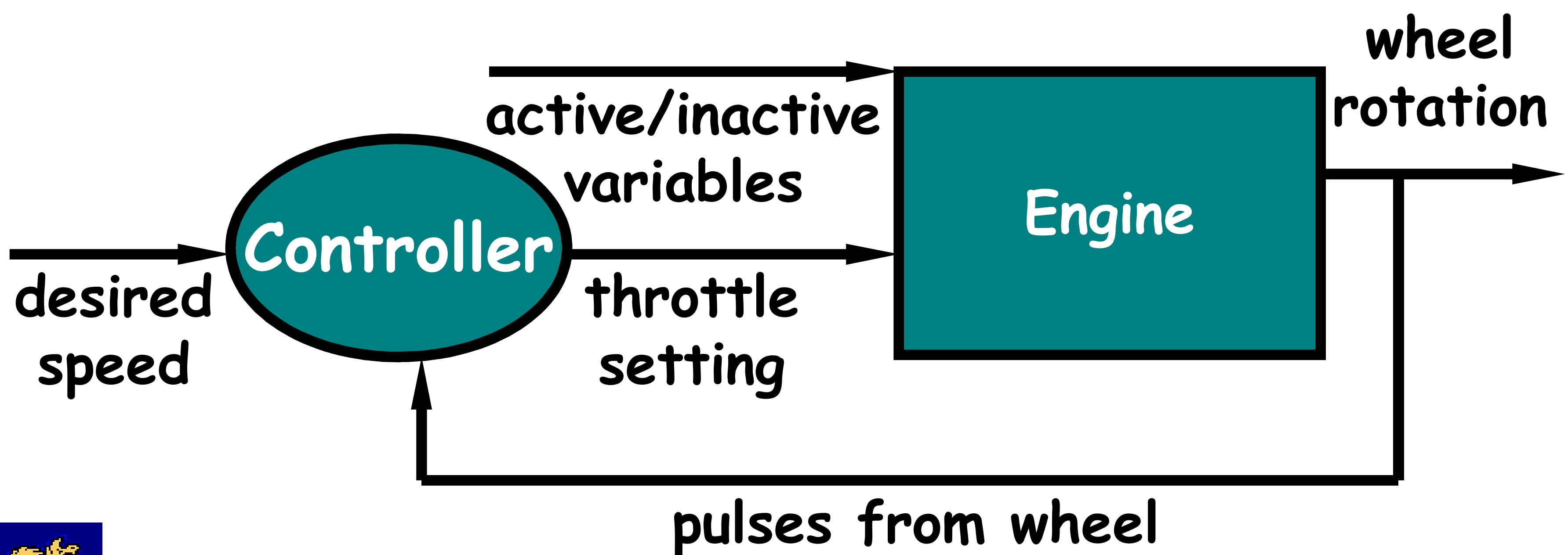
Open-Loop Control System

- Information about process variables is not used to adjust the system.



Process Control Examples

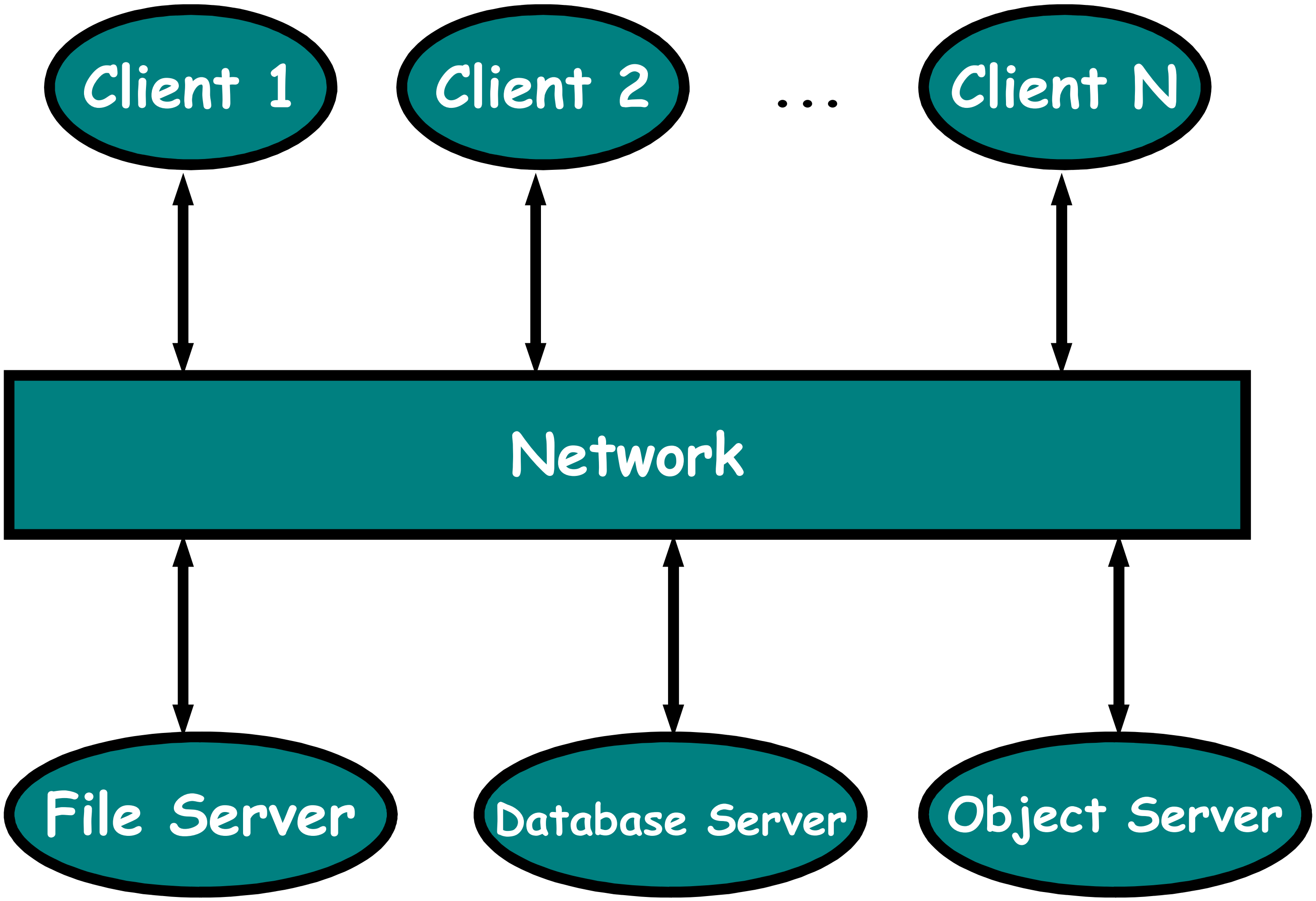
- Real-Time System Software to Control:
  - Automobile Anti-Lock Brakes
  - Nuclear Power Plants
  - Automobile Cruise-Control



Client-Server Style

- Suitable for applications that involve distributed data and processing across a range of components.
- **Components:**
  - **Servers:** Stand-alone components that provide specific services such as printing, data management, etc.
  - **Clients:** Components that call on the services provided by servers.
- **Connector:** The network, which allows clients to access remote servers.

Client-Server Style



Client-Server Style Examples

- **File Servers:**
  - Primitive form of data service.
  - Useful for sharing files across a network.
  - The client passes requests for files over the network to the file server.

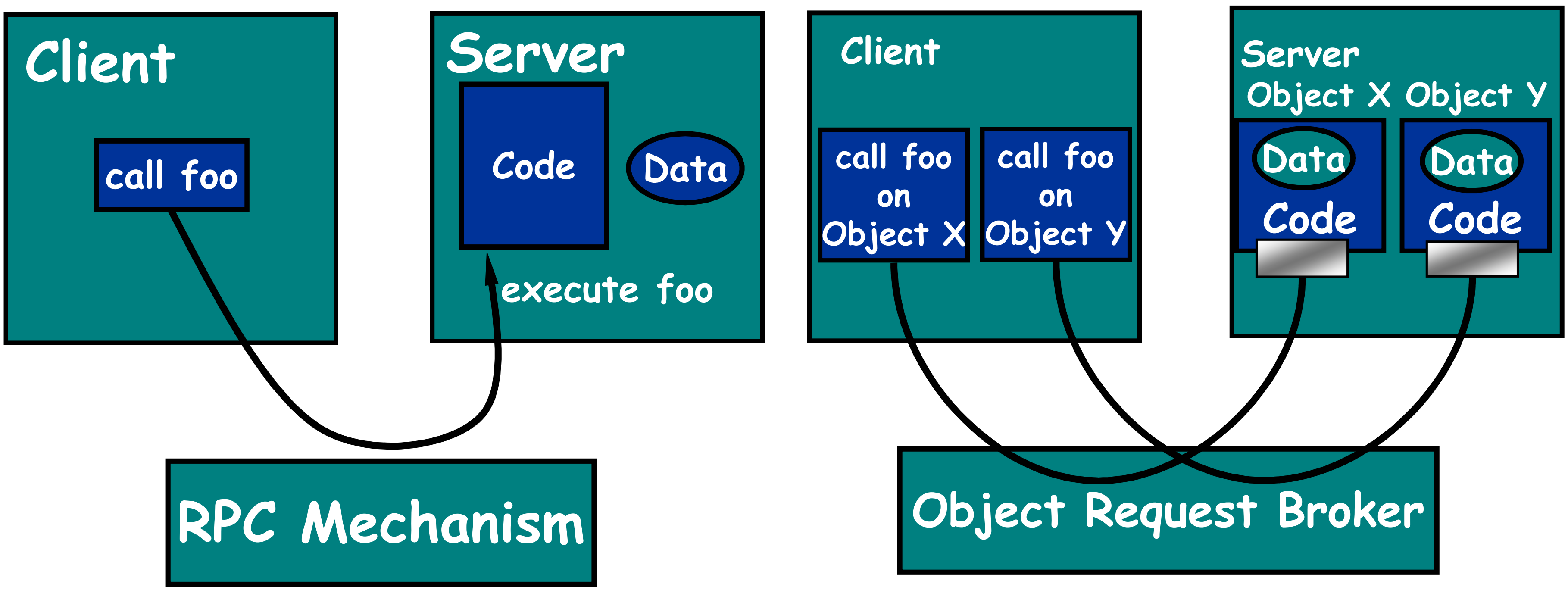
Client-Server Style Examples (Cont'd)

- **Database Servers:**
  - More efficient use of distributing power than file servers.
  - Client passes SQL requests as messages to the DB server; results are returned over the network to the client.
  - Query processing done by the server.
  - No need for large data transfers.
  - Transaction DB servers also available.

Client-Server Style Examples (Cont'd)

- **Object Servers:**
  - Objects work together across machine and network boundaries.
  - ORBs allow objects to communicate with each other across the network.
  - IDLs define interfaces of objects that communicate via the ORB.
  - ORBs are the evolution of the RPC.

RPCs Versus ORBs



1) Remote Procedure Call (RPC)

2) Object Request Broker

Client-Server Advantages

- Straightforward distribution of data.
- Transparency of location.
- Mix and match heterogeneous platforms,
- Easy to add new servers or upgrade existing servers.

Client-Server Disadvantages

- Performance of the system depends on the performance of the network.
- Tricky to design and implement C/S systems.
- Unless there is a central register of names and services, it may be hard to find out what services are available.

Technologies for Distributed Architectures

IBM's MQSeries

- MQSeries provides application-programming services that enable programs to communicate with each other in a distributed fashion using messages and queues.
- This kind of communication is called *asynchronous messaging*.

IBM's MQSeries (Cont'd)

- The MQSeries software enables applications to exchange information across more than 25 different operating system platforms.
- This flexibility allows MQSeries applications to run on hardware ranging from modest desktops to high-end mainframe computers.

MQSeries Components

- *Queue Managers* manage one or more queues and ensure that messages are put on the correct queue or that they are routed to another (remote) queue manager.
- *Applications* must make a successful connection to a queue manager before they can put or get messages to or from a queue.

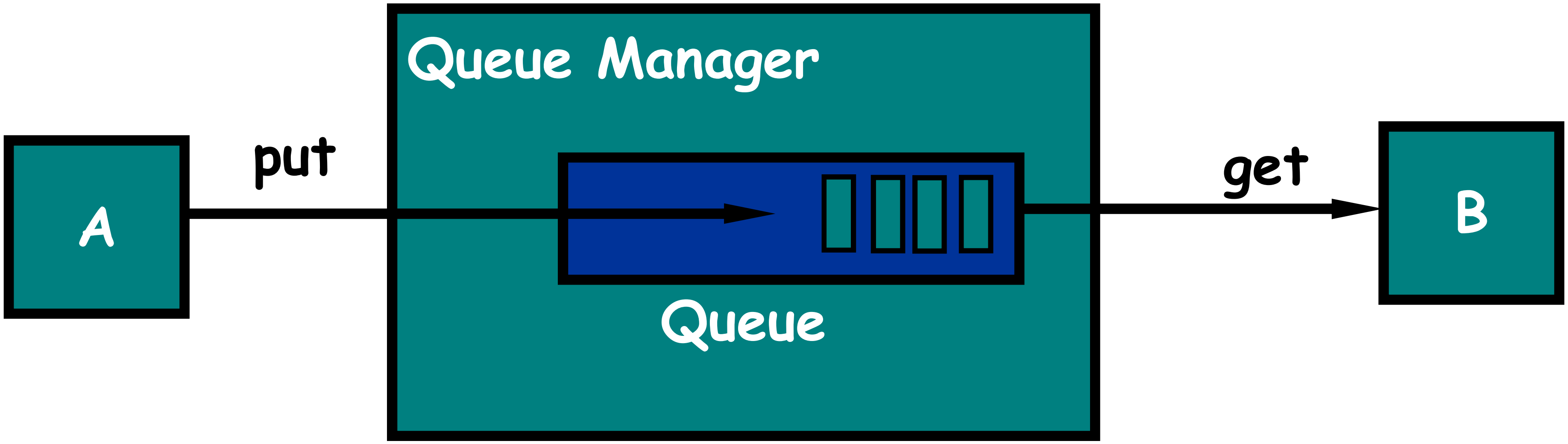
MQSeries Applications

- An application can only connect to one queue manager at a time.
- Before an application can use a queue, it must open a queue for putting, getting, or both putting and getting messages.

MQSeries Queued Messages

- A queued message consists of two parts:
  - The first part includes application-specific data contained in a buffer.
  - The second part includes control information, such as a message type, destination, and various other options.

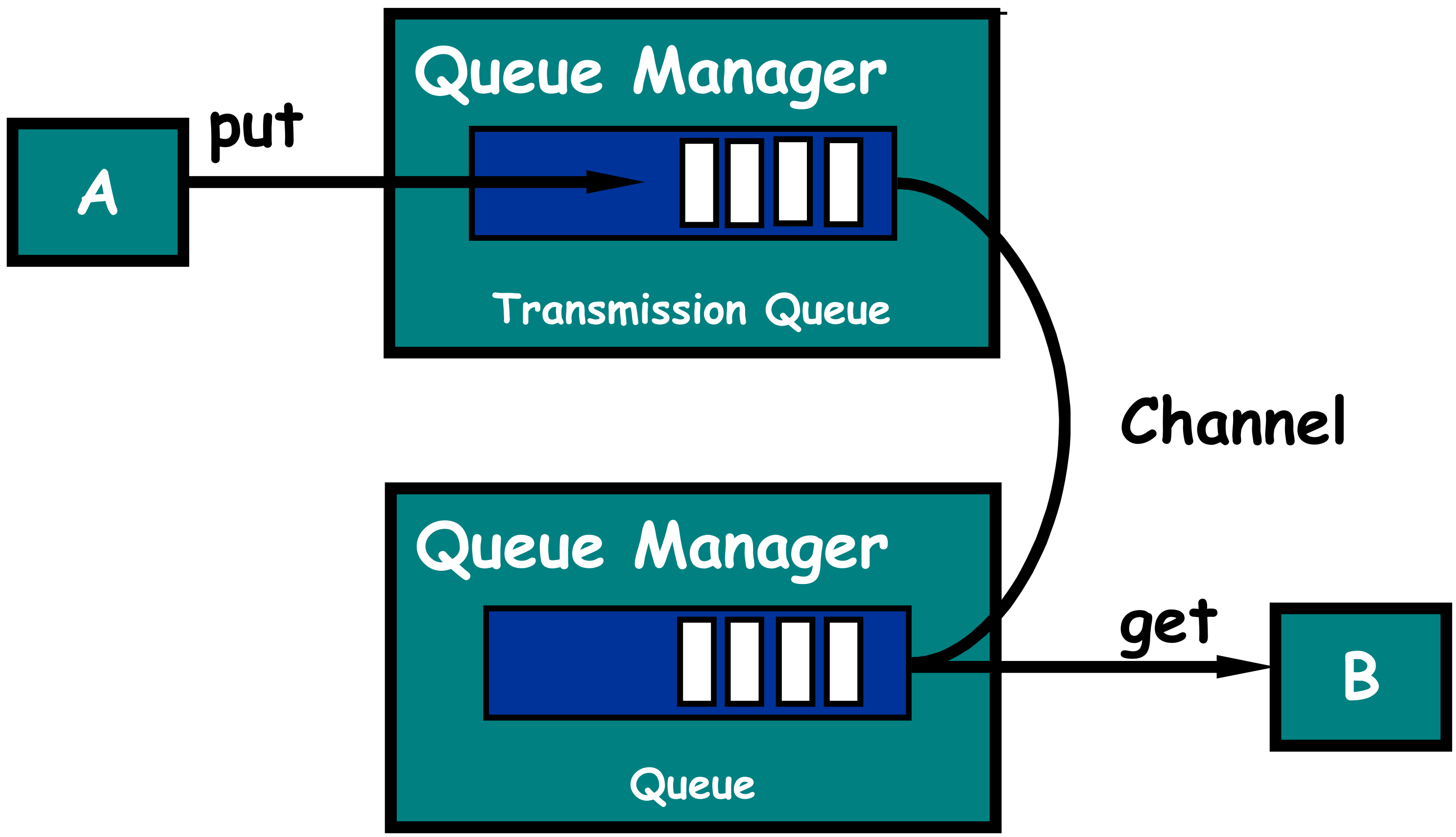
Programs Communicating via a Queue on the Same Workstation



Programs Communicating via a Queue on the Same Workstation

- Figure illustrates two programs **A** and **B** that are communicating through a managed message queue.
- In this example, **A**, **B**, and the **queue manager** are all executing on the same workstation.
- The communication between the programs is conducted through a queue onto which program **A** puts messages and from which program **B** gets messages.

Programs Communicating via a Queue on Different Workstations



Programs Communicating via a Queue on Different Workstations

- Figure illustrates two programs **A** and **B** that are communicating through a managed message queue.
- **A**,**B** are executing on different workstations.
- Program **A** puts a message onto the queue, specifying not a local queue but a local definition of a remote queue.

Programs Communicating via a Queue on Different Workstations

- The local queue definition identifies a non-local queue that is managed by another queue manager.
- The queue manager, to which program **A** is connected to, puts the message on a special queue called a *transmission queue*.
- The message is then automatically sent along a defined channel that connects the two queue managers.

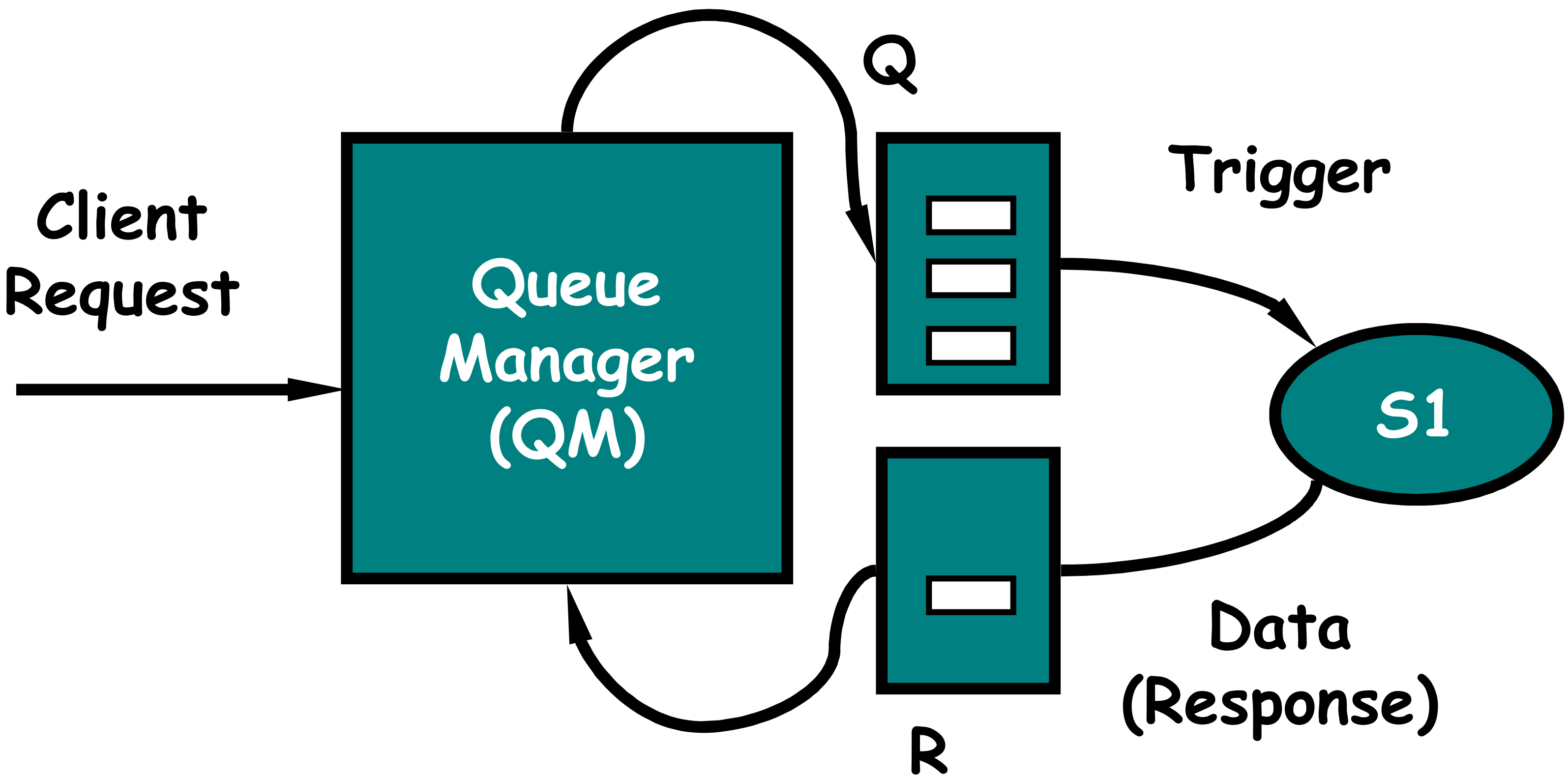
Programs Communicating via a Queue on Different Workstations

- If for some reason the channel is not active (possibly due to a network failure) the message remains on the transmission queue.
- The message will be sent automatically when the channel is available again.

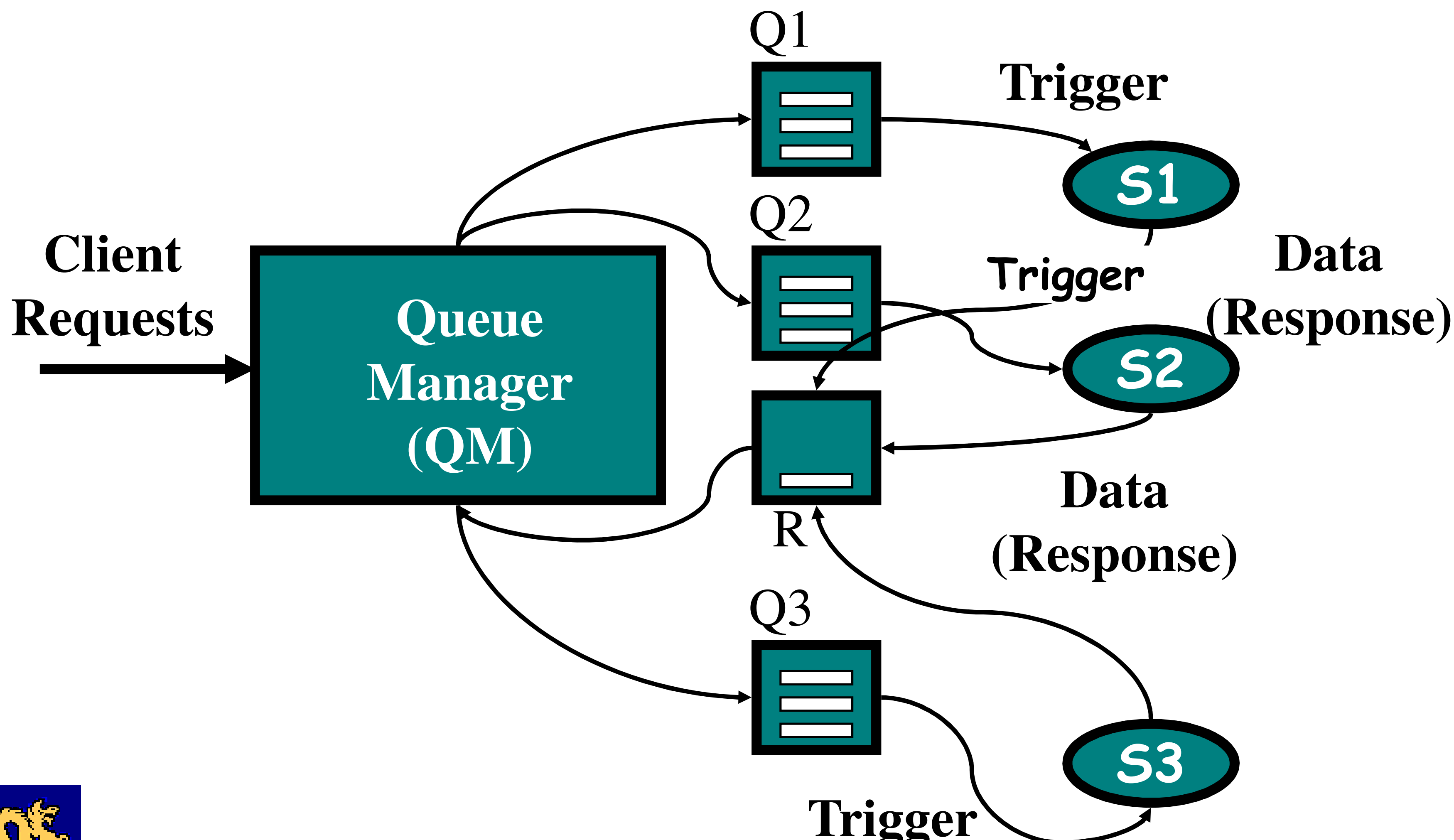
Programs Communicating via a Queue on Different Workstations

- The destination queue manager puts the message on the queue that is specified by program **A**.
- Once a message is placed on the destination queue, the queue manager can invoke program **B** automatically and **B** can then get the queued message.

Using MQSeries to Create a Server that Handles a Single Service S1



Scaling-Up to Multiple Queues and Services



OMG's CORBA

- The Common Object Request Broker Architecture (CORBA) is a standard distributed object architecture developed by the Object Management Group (OMG) consortium.

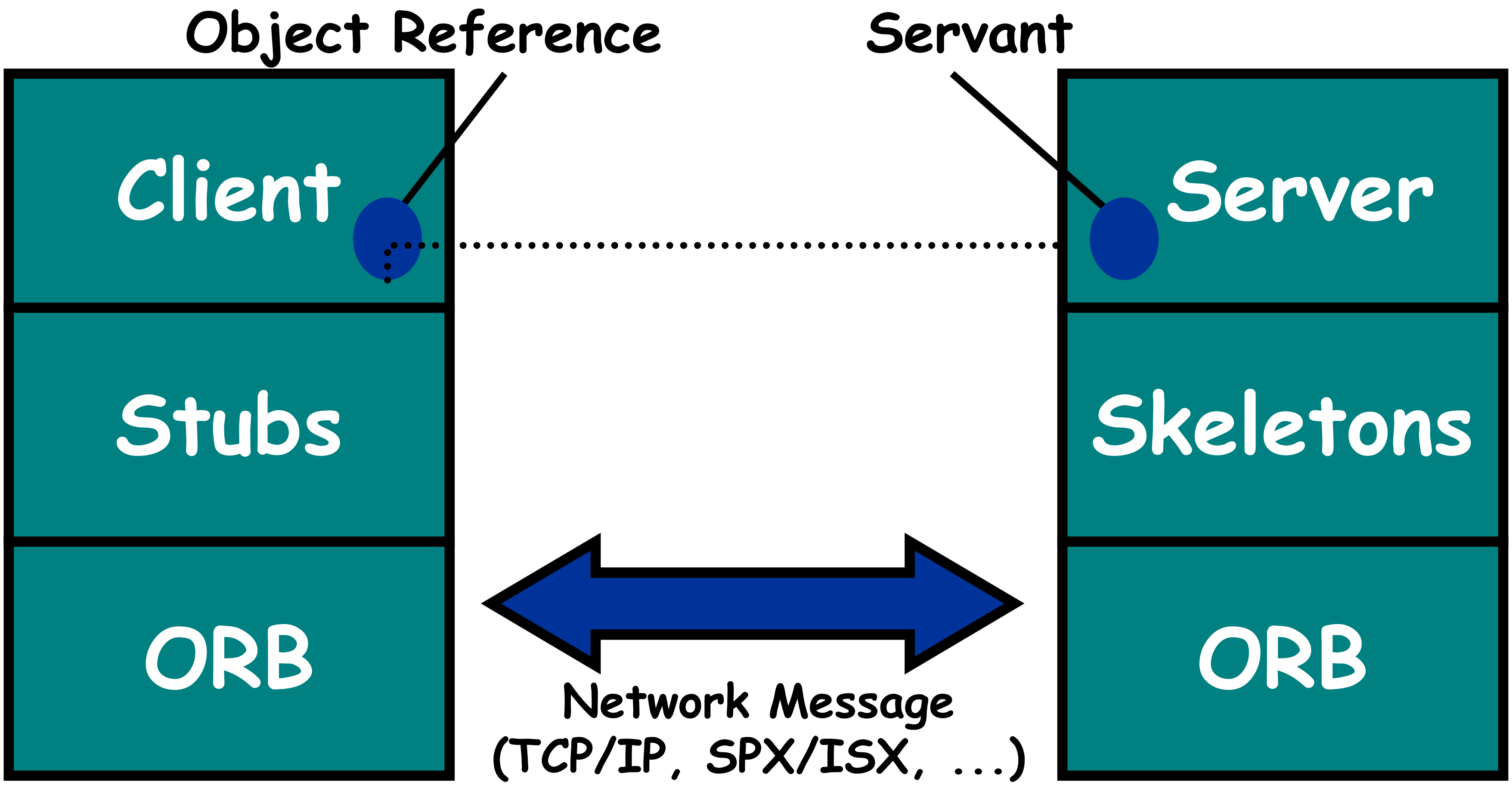
CORBA Objects

- CORBA objects can:
  - be located anywhere on the network,
  - interoperate with objects on other platforms,
  - be written in a variety of programming languages:
    - Java
    - C++
    - C
    - Smalltalk
    - COBOL
    - Ada.

CORBA Messages

- Distributed objects in a CORBA system communicate by sending messages to each other.
- These messages, however, are not queued, as is the case with MQSeries.

CORBA Method Request



Method Requests

CORBA Method Request (Cont'd)

- The Figure shows how a message from a client object is sent to a server object.
- In order for a client to access a remote server object, it must first obtain a handle (object reference) to that object.
- If the server object is remote, the handle points to a stub function, which uses the Object Request Broker (ORB) service to forward invocations to the server object.

CORBA Stubs

- After the stub establishes a connection to the server, it sends the following to the destination object:
  – an object reference,
  – an encoded representation of the method,
  – parameters to the skeleton code linked.

CORBA Skeletons

- The skeleton code transforms the call and parameters into the required implementation-specific format before calling the object.

CORBA Platform Independence

- The client is unaware of the CORBA object's location, implementation details, and which ORB is used to access the object.
- The connections between distributed objects are managed through a name server.

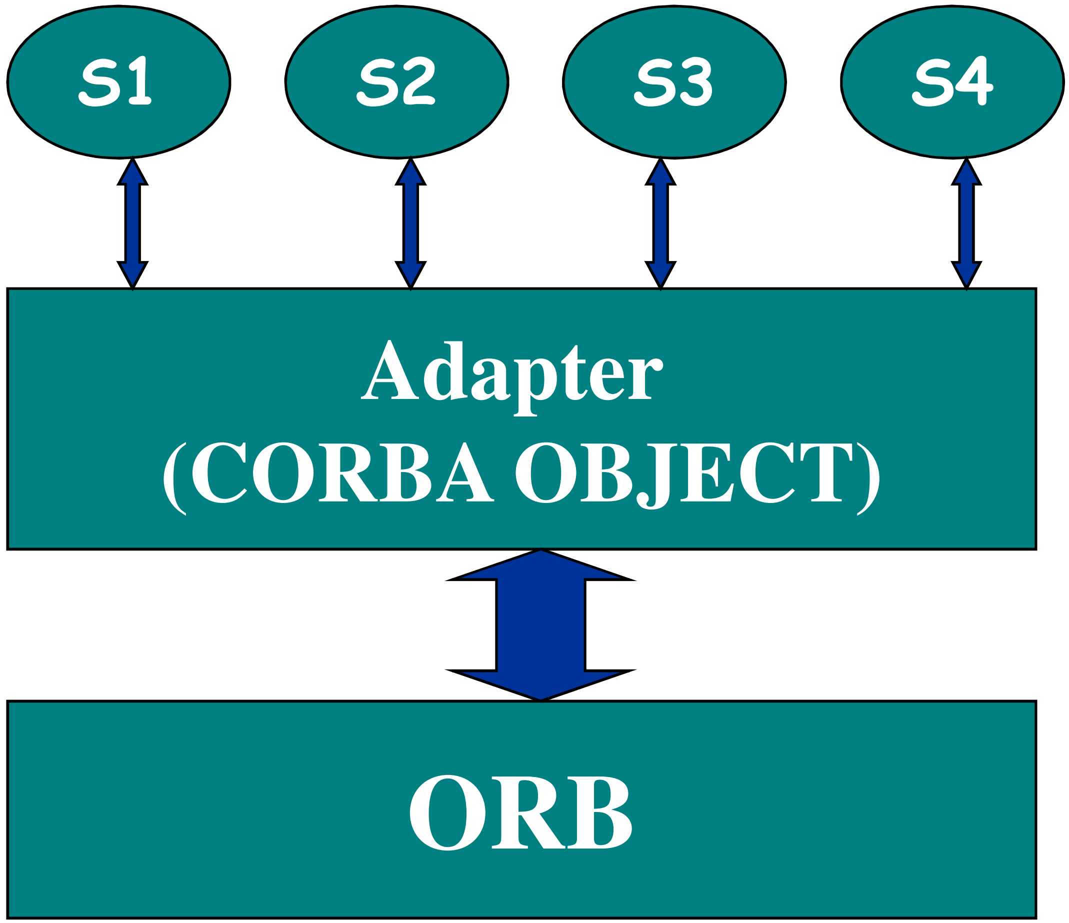
CORBA IDL and IIOP

- The client may only invoke methods that are specified in the CORBA object's interface.
- Object interfaces are defined using OMG's Interface Definition Language (IDL).
- Different ORBs communicate via the Internet InterORB Protocol (IIOP).

Server Objects in CORBA

- The server side ORB receives the request over a network connection and then determines which of the objects on its machine is the target.
- When the ORB locates the object, it must prepare it to receive the request. *E.g.*,
  - Start a server process that contains the object.
  - Retrieve the object from persistent storage.

Hiding Services Behind an Object Adapter



Object Adapters in CORBA

- The Figure shows how an adapter can act as a proxy between a set of services and the ORB.
- Clients will access each service through the adapter that is responsible for that service.
- The adapter will be responsible for finding the appropriate filters to handle each client request.

Object Adapters in CORBA (Cont'd)

- These filters may be:
  - on the same machine as the adapter,
  - or may be on another machine, in which case the adapter must delegate the client request to another adapter.

References

- [Bass et al 98] Bass, L., Clements, P., Kazman R., Software Architecture in Practice. SEI Series in Software Engineering, Addison-Wesley, 1998
- [CORBA98] CORBA. 1998. OMG's CORBA Web Page. In: http://www.corba.org
- [Gamma95] Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Inc., Reading, Massachusetts.
- [IBM98] MQSeries Whitepaper. In: http://www.software.ibm.com/ts/mqseries/library/whitepapers/mqover
- [JavaIDL98] Lewis, G., Barber, S., Seigel, E. 1998. Programming with Java IDL: Developing Web Applications with Java and CORBA. Wiley Computer Publishing, New York.
- [CORBA96] Seigel, J. 1996. CORBA Fundamentals and Programming. John Wiley and Sons Publishing, New York.
- [Shaw96] Shaw, M., Garlan, D. 1996. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall.