National University
Of Computer and Emerging Sciences

Software Design and Architecture

Architectural Styles and Patterns

Sajid Anwer

Department of Software Engineering,
FAST-NUCES, CFD Campus

Lecture Outline

- Fundamentals of Architectural Styles and Patterns
- Different types of Architectural Patterns
- Different types of Architectural Styles

Lecture Material

- Software Architecture, Foundation, Theory, and Practice (Ch#3, Ch#4, and Ch#11)

Fundamentals of Architectural Styles

- In some scenarios, certain design decisions results in better solution as compared to other alternatives.
- Example
  - » Physically *separate the software components* used to request services from the components that provide the needed services.
  - » Make the service provider *unaware of the requesters*' identify to allow the providers to service transparently many, possibly changing requesters.
  - » *Insulate the requesters* from one another to allow for their independent addition, removal and modification.
  - » Allow for multiple service providers to *emerge dynamically* to off-load the existing providers should the demand for service increases above a given threshold.

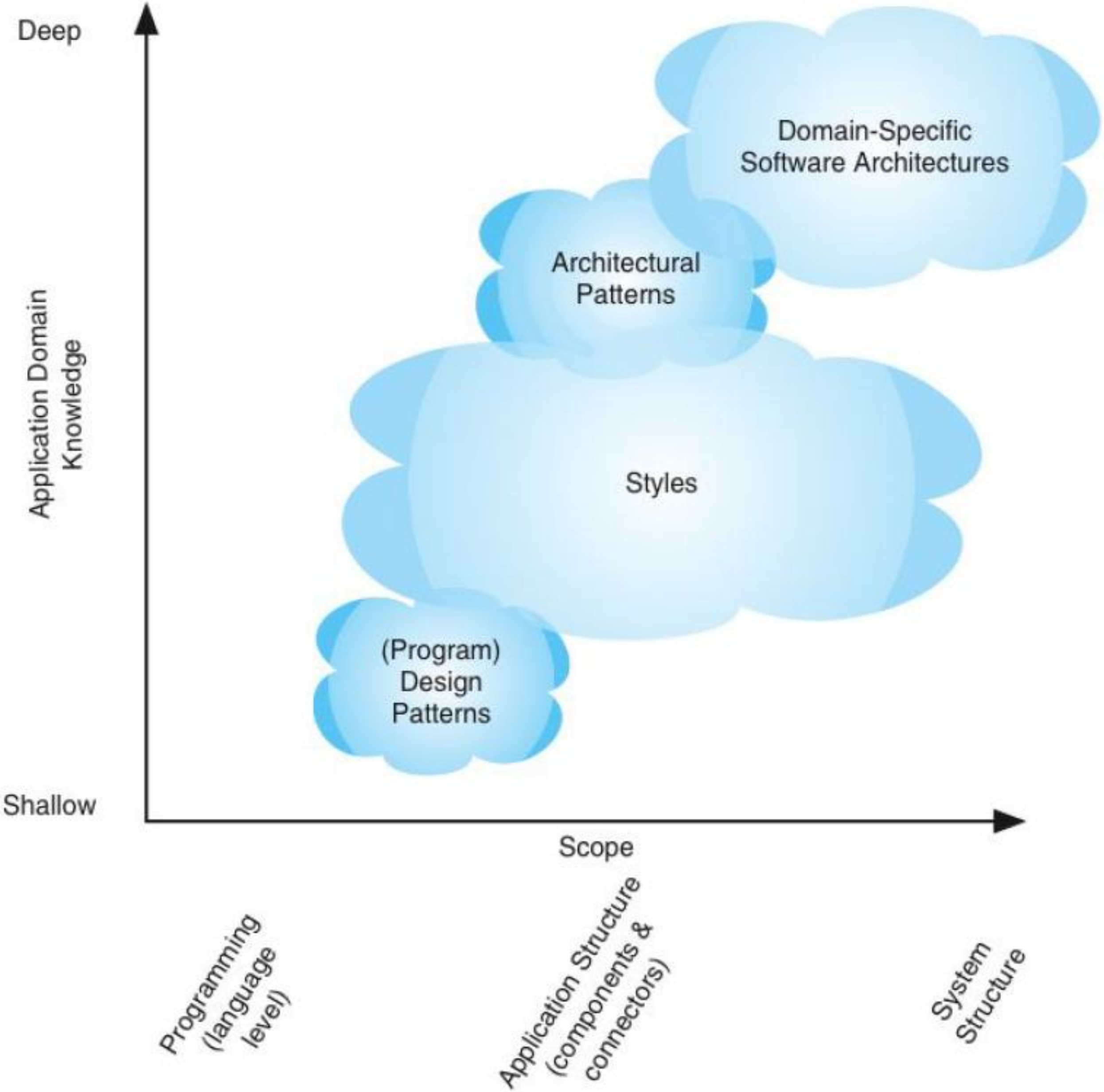
Fundamentals of Architectural Styles

- An *architectural style* is a *named collection* of architectural design decisions that
  - are applicable in a given development context
  - constrain architectural design decisions that are specific to a particular system within that context
  - elicit beneficial qualities in each resulting system

Fundamentals of Architectural Patterns

- An *architectural pattern* is a set of architectural design decisions that are applicable to a *recurring design problem*, and parameterized to account for different software development contexts in which that problem appears.
  - » Science
  - » Banking
  - » E-commerce
  - » Reservation systems

Architectural Styles vs Patterns

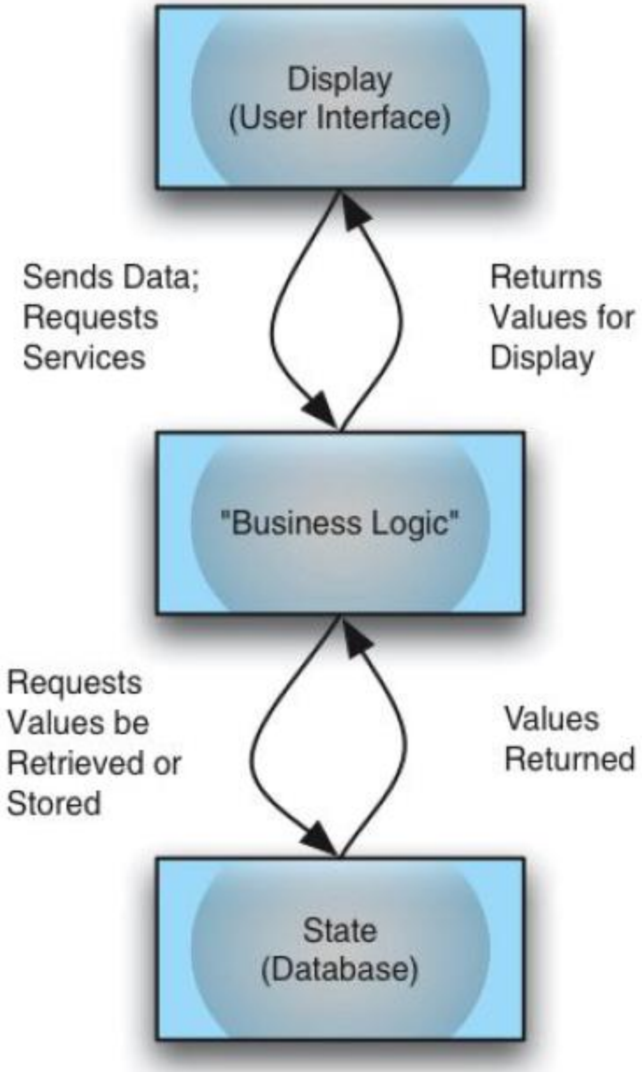

Deep
Application Domain Knowledge
Shallow
Domain-Specific Software Architectures
Architectural Patterns
Styles
(Program) Design Patterns
Scope
Programming (language level)
Application Structure (components & connectors)
System Structure

Architectural Styles vs Patterns

- An *architectural Styles and patterns* can be compared in context of
  - » Scope
    - An architectural style applies to a development context (e.g. highly distributed systems, GUI intensive).
    - An architectural pattern applies to a specific design problem (e.g. the systems' state must be presented in multiple ways)
  - » Abstraction
  - » Relationship
    - A system designed according to the rules of a single style may involve the use of multiple patterns.
    - A single pattern could be applied to systems designed according to the guidelines of multiple styles.

Architectural Patterns – State-Logic-Display

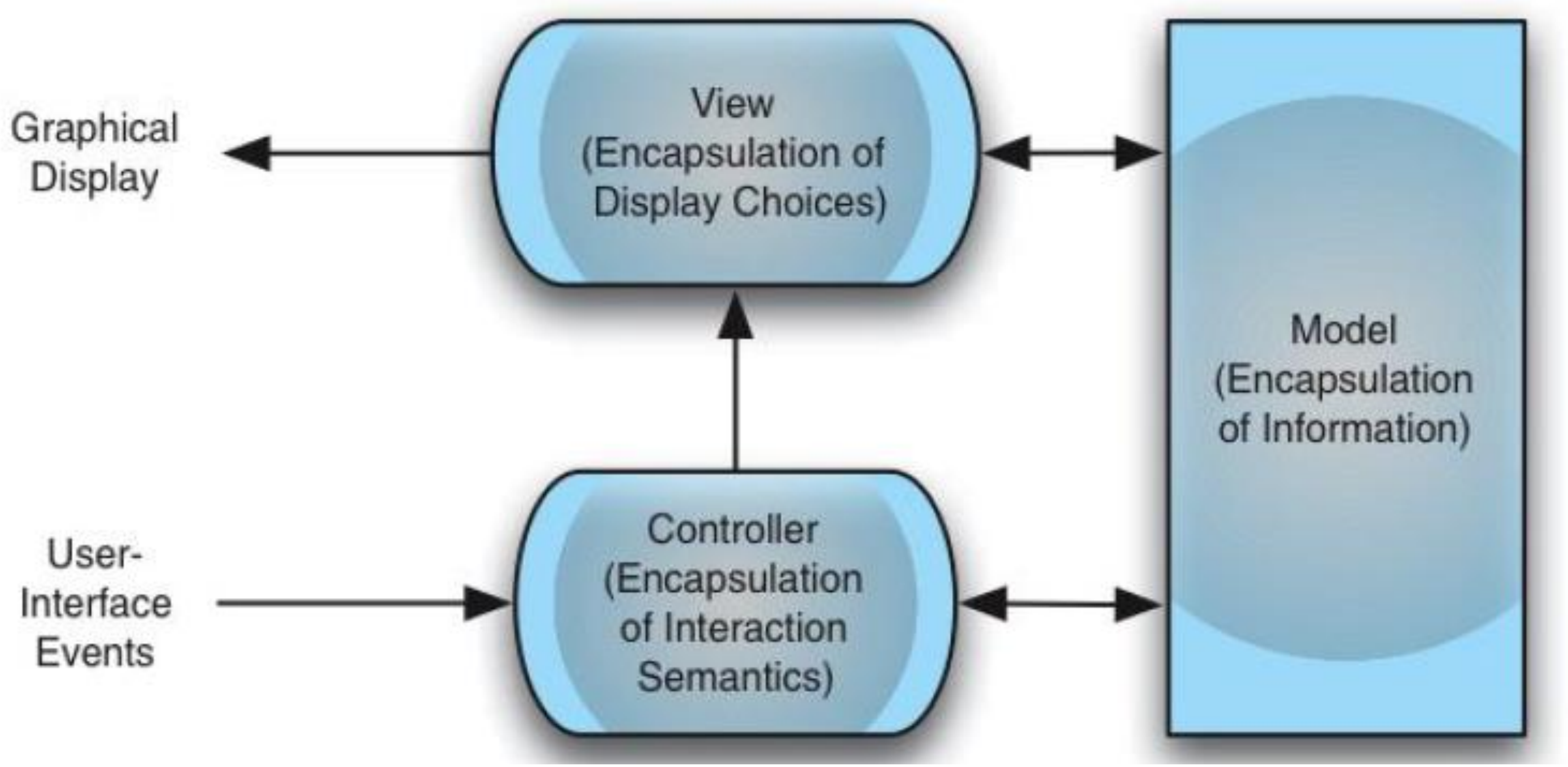
- Business applications
- Multiplayer games
- Web applications


Display
(User Interface)
Sends Data; Requests Services
Returns Values for Display
"Business Logic"
Requests Values be Retrieved or Stored
Values Returned
State
(Database)

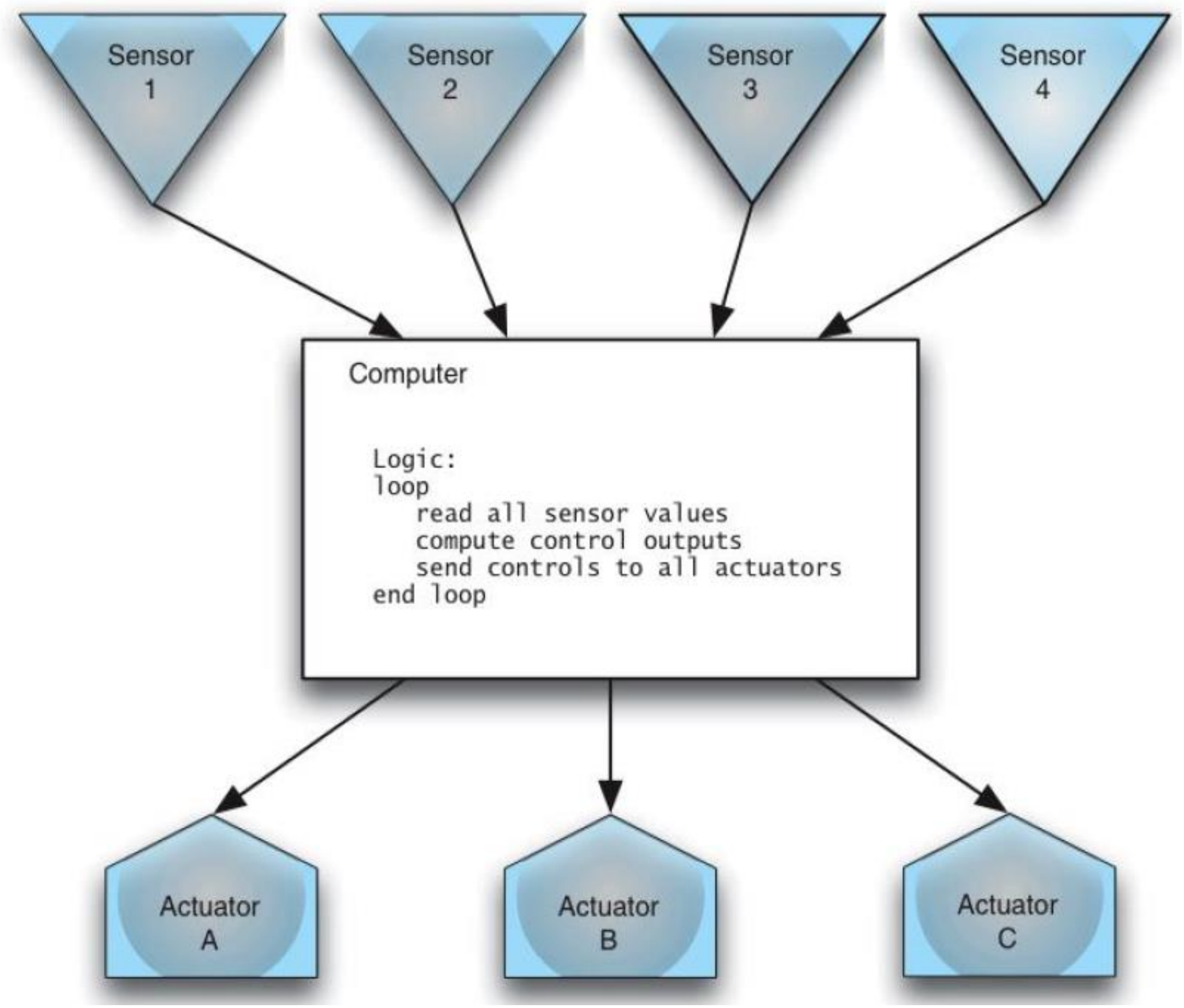
Architectural Patterns – Model-View-Controller (MVC)

- Objective: *Separation* between information, presentation and user interaction.
- When a model object value changes,
  - » A notification is sent to the view and to the controller.
  - » The view can update itself and the controller can modify the view if its logic so requires.
- When handling input from the user the Windows system sends the user event to the controller;
  - » If a change is required, the controller updates the model object.

Architectural Patterns – Model-View-Controller (MVC)

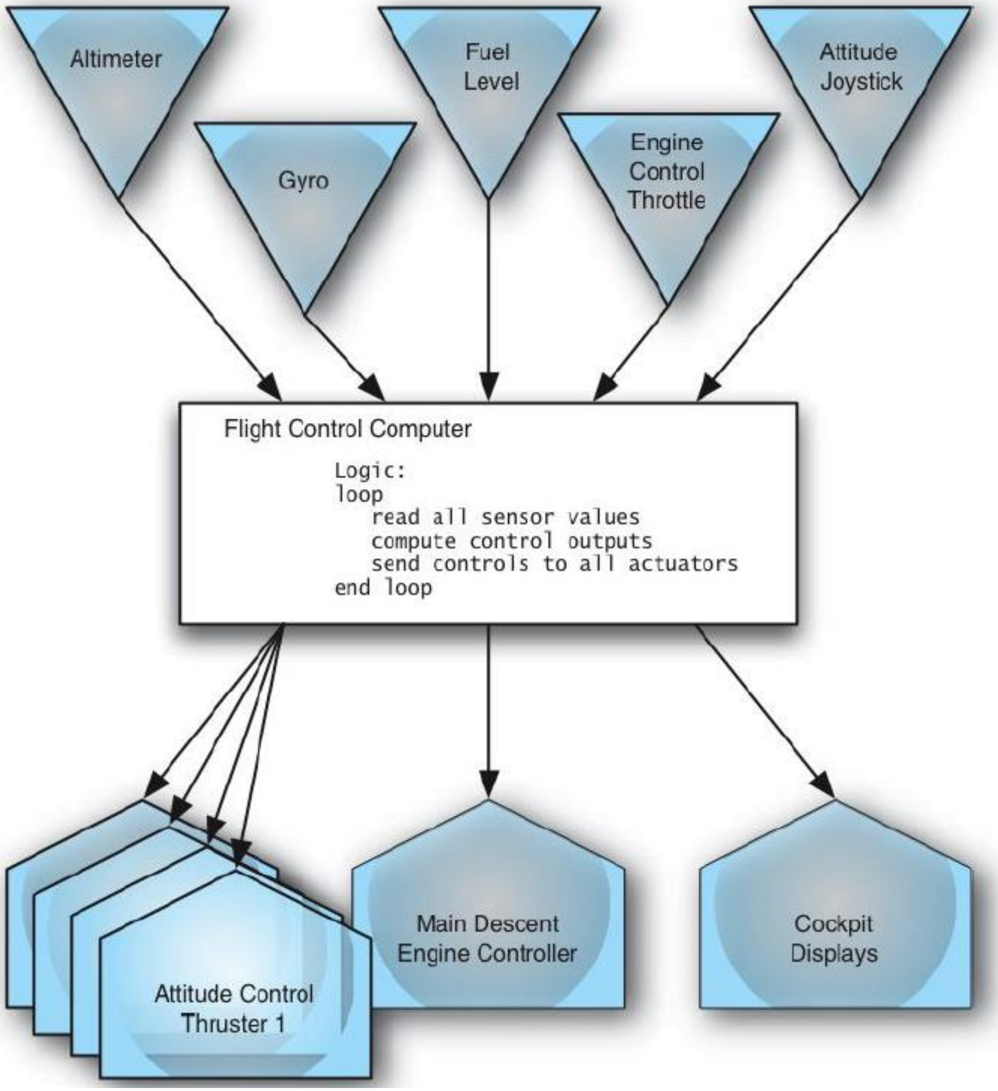

Graphical Display
View (Encapsulation of Display Choices)
Model (Encapsulation of Information)
User-Interface Events
Controller (Encapsulation of Interaction Semantics)

Architectural Patterns – Sense-Compute-Control (SCC)



```
Logic:
loop
   read all sensor values
   compute control outputs
   send controls to all actuators
end loop
```

Architectural Patterns – Sense-Compute-Control (SCC)



```
Logic:
loop
   read all sensor values
   compute control outputs
   send controls to all actuators
end loop
```

Architectural Styles Fundamentals

- A *vocabulary* of design elements
  - Component and connector types; data elements
  - e.g., pipes, filters, objects, servers
- A set of 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

Architectural Styles Fundamentals -- Benefits

- 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
- Interoperability
  - » Supported by style standardization

Architectural Styles Fundamentals -- Types

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

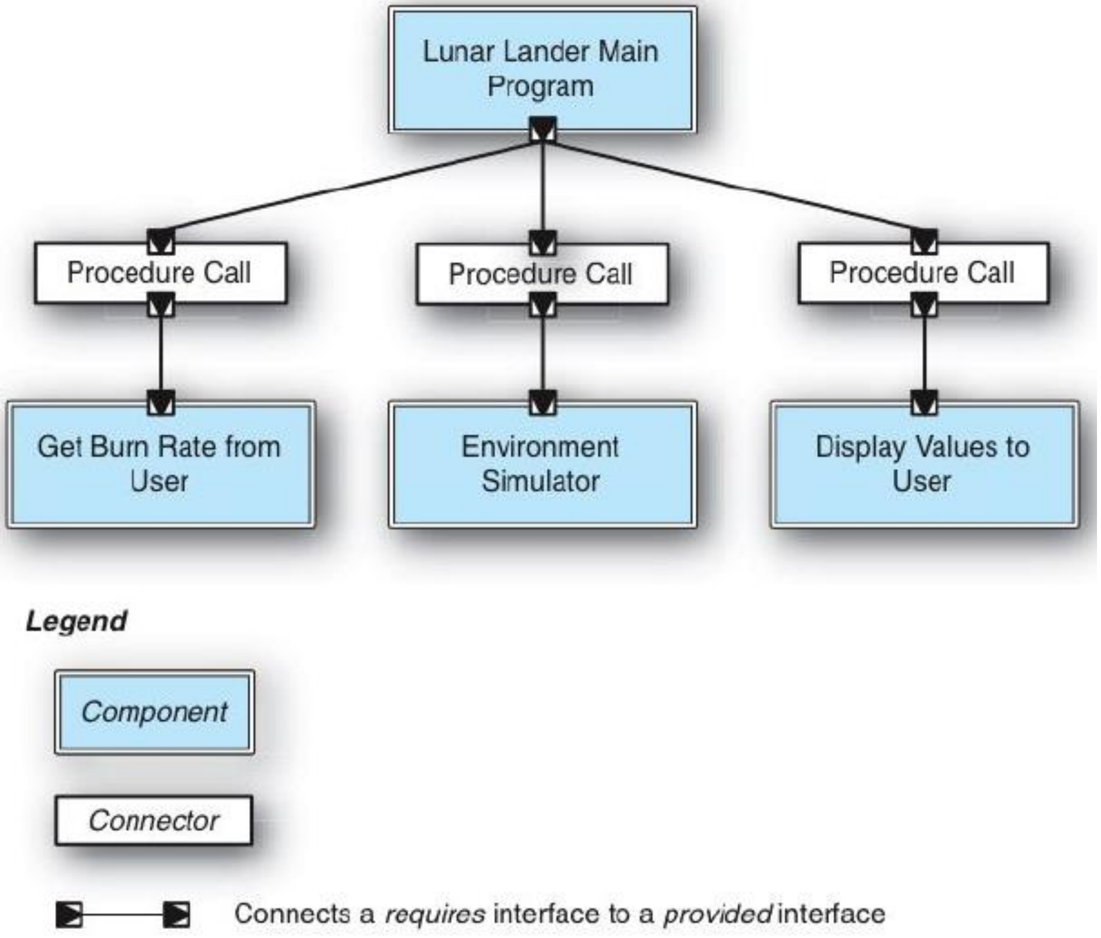
- “Derived” styles
  - » C2
  - » CORBA

Architectural Styles -- Traditional

- Main program and subroutines
  - » Instantly familiar to anyone who has programmed in a language such as C.
  - » Decomposition based on separation of functional processing steps.
  - » **Components** – Mian program and subroutines
  - » **Connectors** – Function/procedure calls
  - » **Data elements** – Values passed in/out of subroutines
  - » **Topology** – Static organization of components is hierarchical; full structure is a directed graph.
  - » **Typical Use**: Small programs
  - » **Relation to programming language** – Traditional imperative programming languages such as BASIC, PASCAL or C.

Architectural Styles -- Traditional

- Main program and subroutines

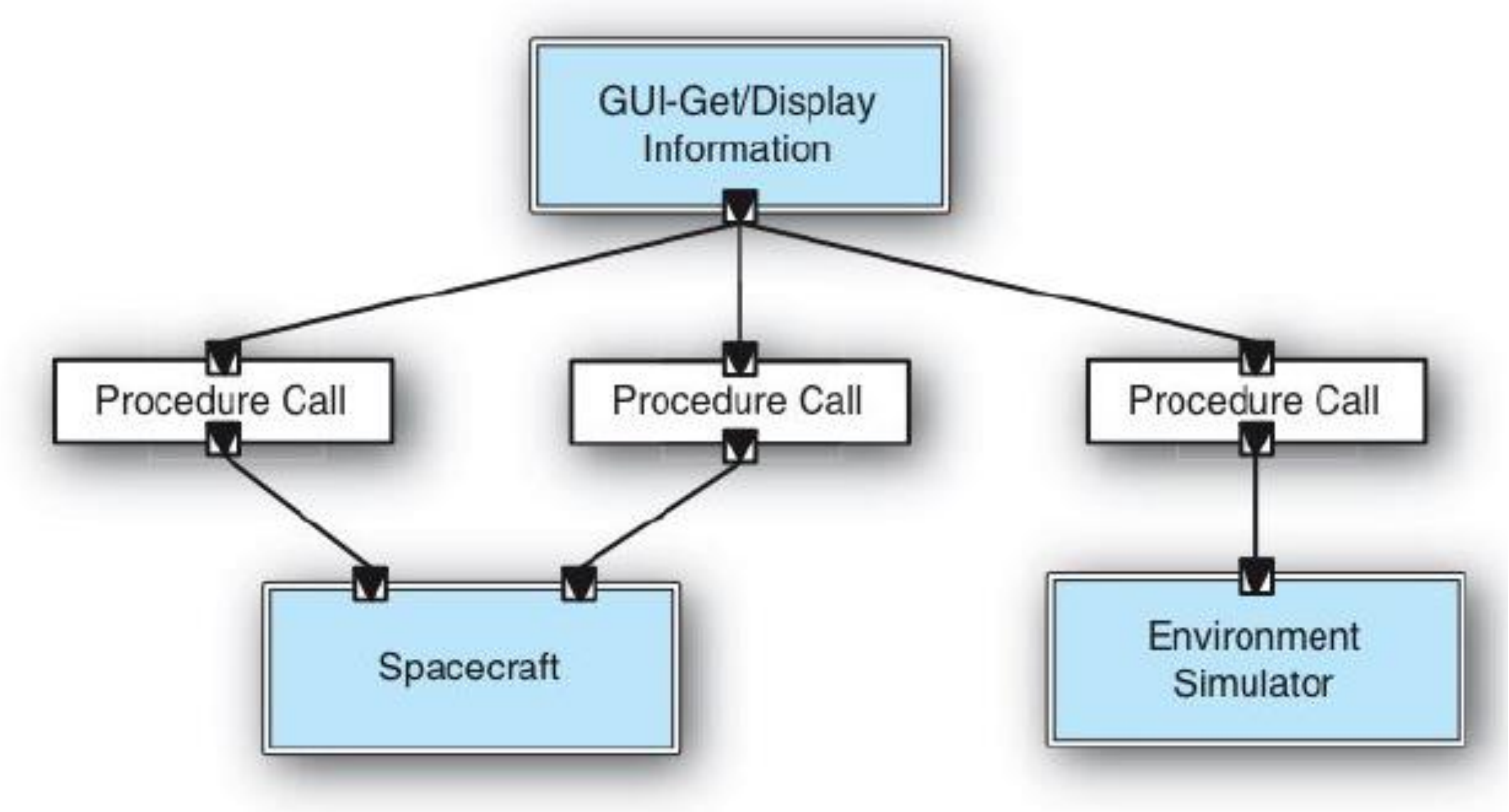


Architectural Styles -- Traditional

- Object Oriented
  - **Components** – Objects (instances of a class)
  - **Connector** – Method invocation (procedure call to manipulate state).
  - **Data Elements** – Arguments to methods.
  - **Topology** – Can vary, components may share data and interface functions through inheritance hierarchies.
  - **Additional Constraints** – Commonly shared memory, single threaded.
  - **Typical Use** – Applications where the designer wants a close correlation between entities in the physical world and entities in the program.
  - **Relation to Programming Language** – Java, C++ etc.

Architectural Styles -- Traditional

- Object Oriented



Architectural Styles -- Layered

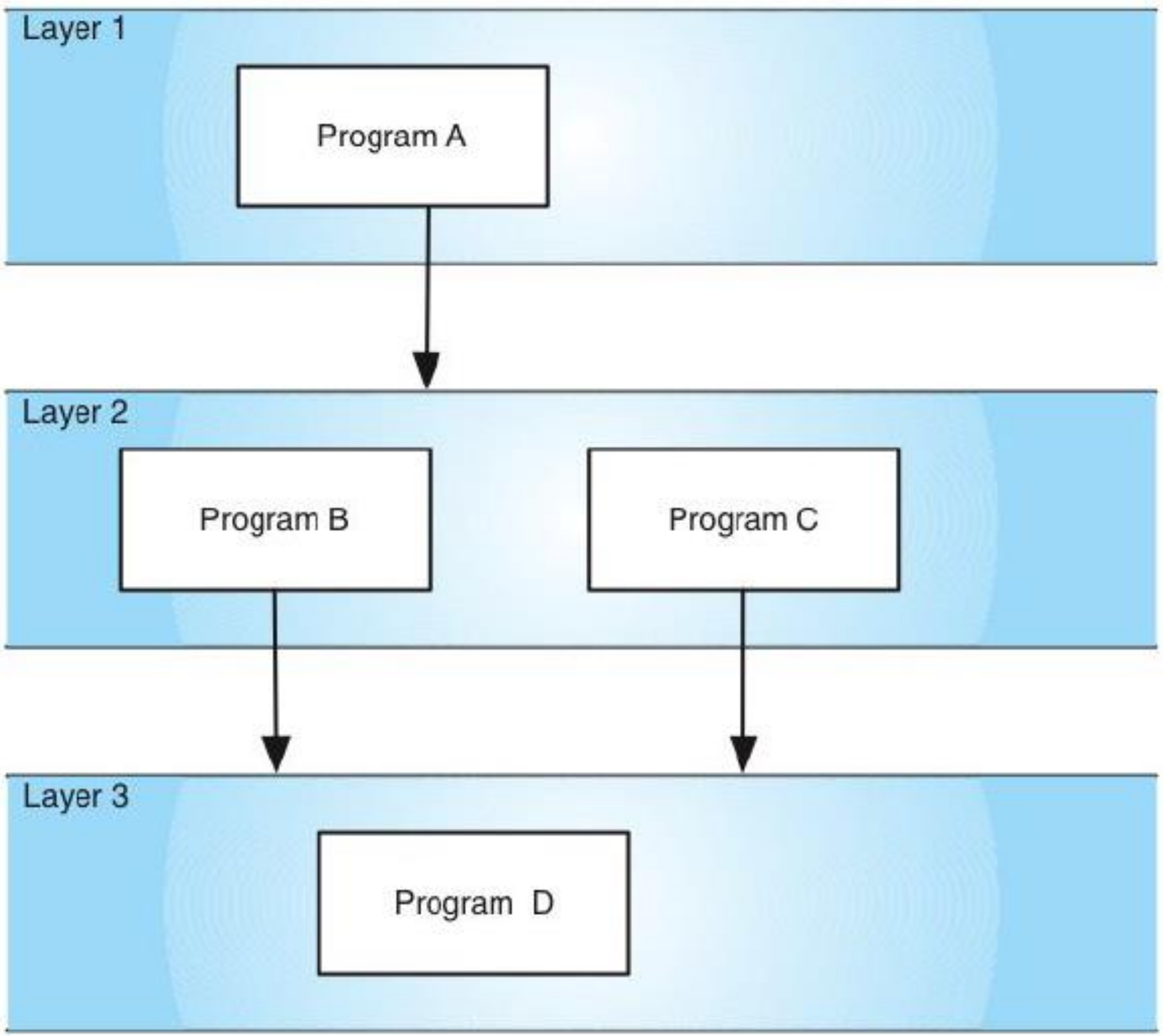
- Consists of ordered sequence of layers;
  - Each layer or virtual machine, offers a *set of services* that may be accessed by programs (subcomponents) residing within the layer above it.

- **Components** – Layers offering a set of services to other layers, typically comprising several programs (subcomponents)
- **Connectors** – Typically procedure calls.
- **Data Elements** – Parameters passed between layers.
- **Topology** – Linear, for strict virtual machines, a directed acyclic graph in looser interpretations.

- **Typical Uses** – Operating System design, network protocol stacks.

- **Cautions** – Strict virtual machine with many levels can be relatively inefficient.

Architectural Styles -- Layered

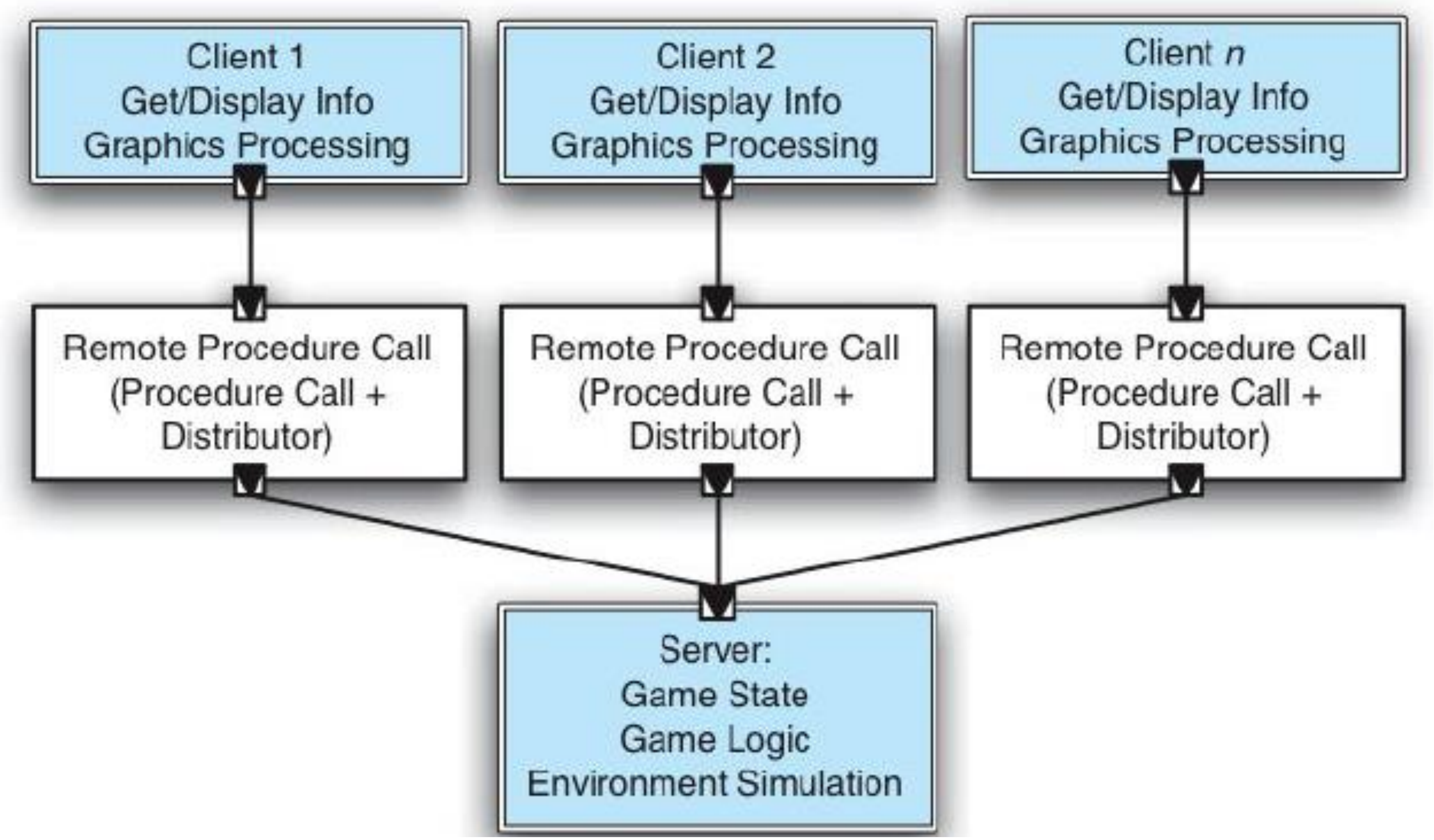


Architectural Styles -- Layered

- Client-Server
  - **Components** – Clients and servers
  - **Connectors** – Remote procedure calls, network protocols
  - **Data Elements** – Parameters and return values as sent by connectors
  - **Topology** – Two level, with multiple clients making requests to the server.
  - **Typical Use**
    - Applications where centralization of data is required.

Architectural Styles -- Layered

- Client-Server



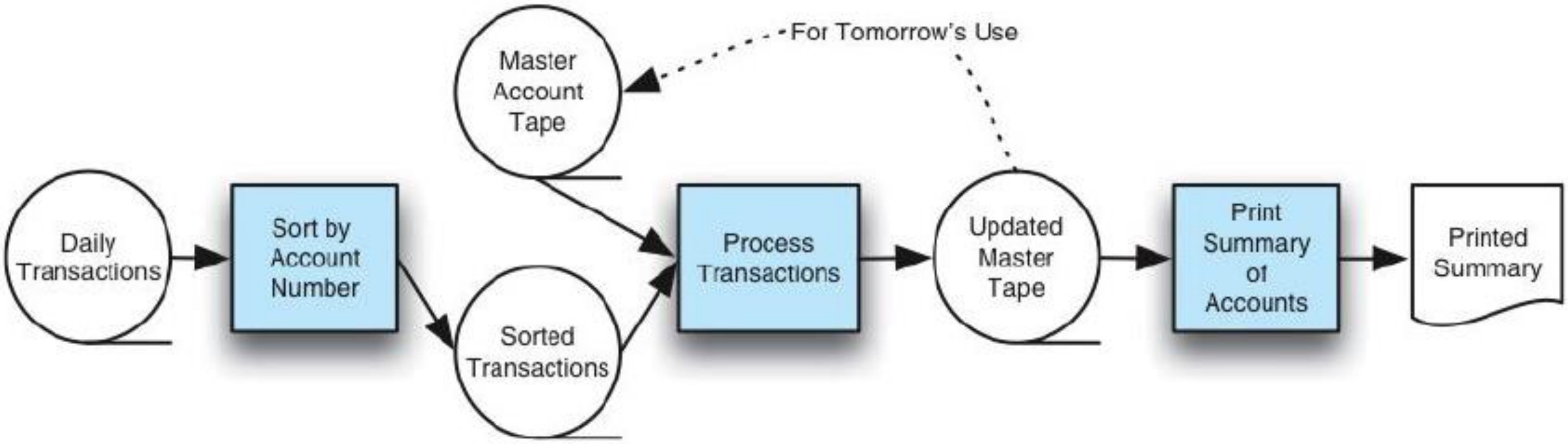
Architectural Styles – Data Flow

- Batch Sequential
  - Separate programs are executed in order;
    - Data is passed an aggregate from one program to the next.
  - **Components** – Independent programs.
  - **Connectors** – The human hand carrying tapes between the programs.
  - **Data elements** – Explicit, aggregate elements passed from one component to the next upon completion of the producing program’s execution.
  - **Topology** – Linear.
  - **Additional Constraints** – One program runs at a time, to completion.
  - **Typical Use** – Transaction processing in financial systems.
  - **Relation to programming language** – None

  - **Cautions** – When interaction between the components is required; when concurrency between components is possible or required.

Architectural Styles – Data Flow

- Batch Sequential



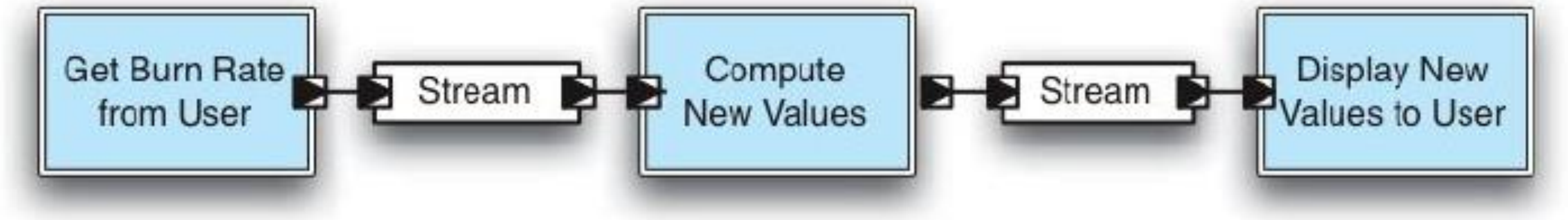
Architectural Styles – Data Flow

- Pipe-and-Filter
  - Separate programs are executed, potentially concurrently; data is passed as a stream from one program to the next.

  - **Components** – Independent programs, known as filters
  - **Connectors** – Explicit routers of data streams; service provided by operating system.
  - **Data Elements** – Not explicit; must be (linear) data stream.
  - **Topology** – Pipeline
  - **Qualities Yielded** – Filters are mutually independent. Simple structure of incoming and outgoing data stream facilitates novel combinations of filters for new, composed applications.
  - **Typical Use** – Ubiquitous in operating system application programming.
  - **Relation to programming language** – Prevalent in Unix Shell.

Architectural Styles – Data Flow

- Pipe-and-Filter

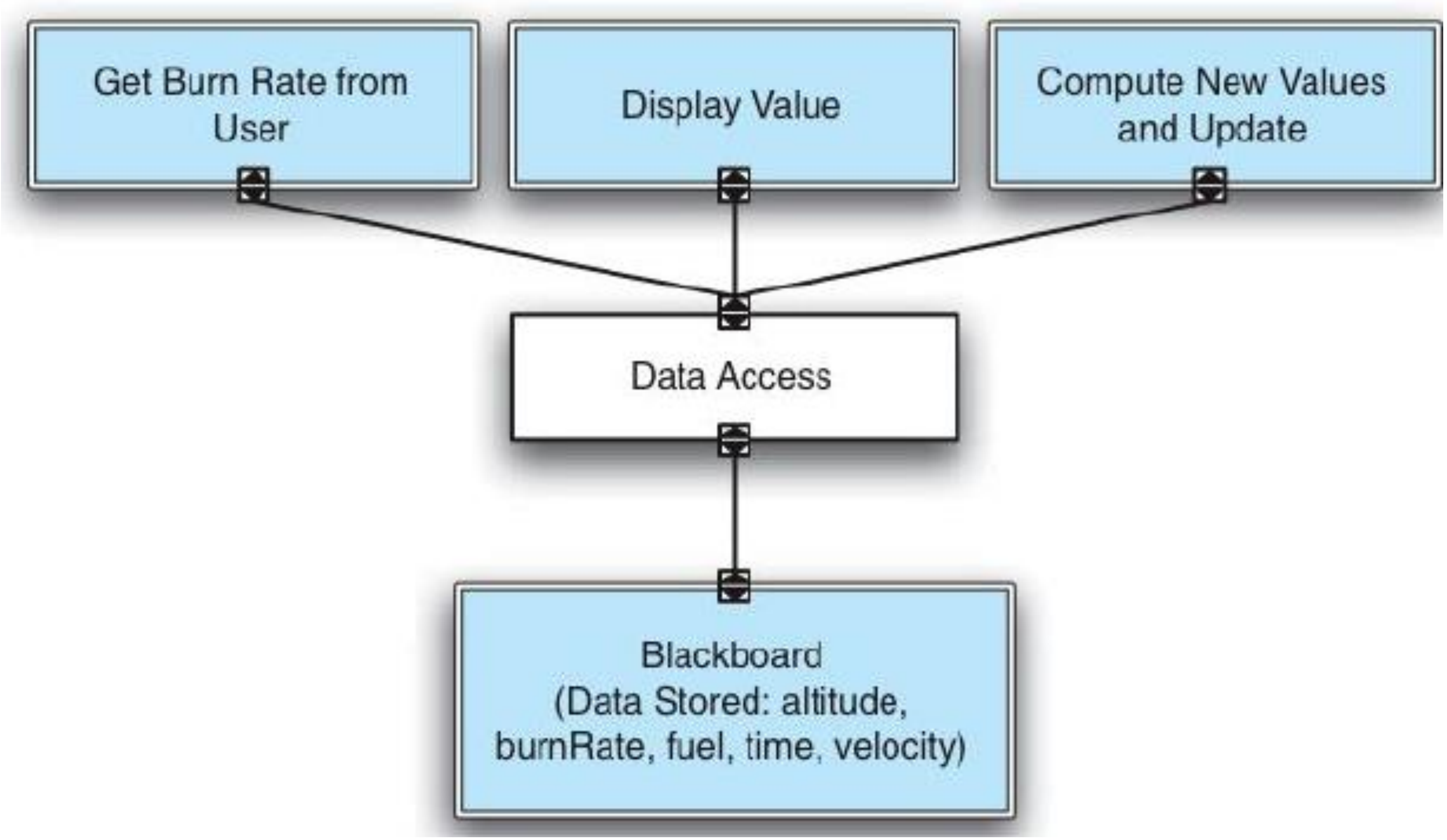


Architectural Styles – Shared Memory

- Blackboard
  - Independent programs access and communicate *exclusively* through a *global data repository*, known as a blackboard.
  - **Components** – Independent programs, sometimes referred to as 'knowledge sources'
  - **Connects** – Access to the blackboard may be by direct memory reference or can be through a procedure call or a database query.
  - **Data Elements** – Data stored in the blackboard
  - **Topology** – Star topology, with the blackboard at the center.
  - Variants – In one version of the style,
    - Programs poll the blackboard to *determine* if any values of interest have changed;
    - In another version, a blackboard manager *notifies* interested components of an update to the blackboard.
  - Typical Use – Heuristic problem solving in artificial intelligence applications

Architectural Styles – Shared Memory

- Blackboard


Get Burn Rate from User
Display Value
Compute New Values and Update
Data Access
Blackboard
(Data Stored: altitude, burnRate, fuel, time, velocity)

Architectural Styles – Shared Memory

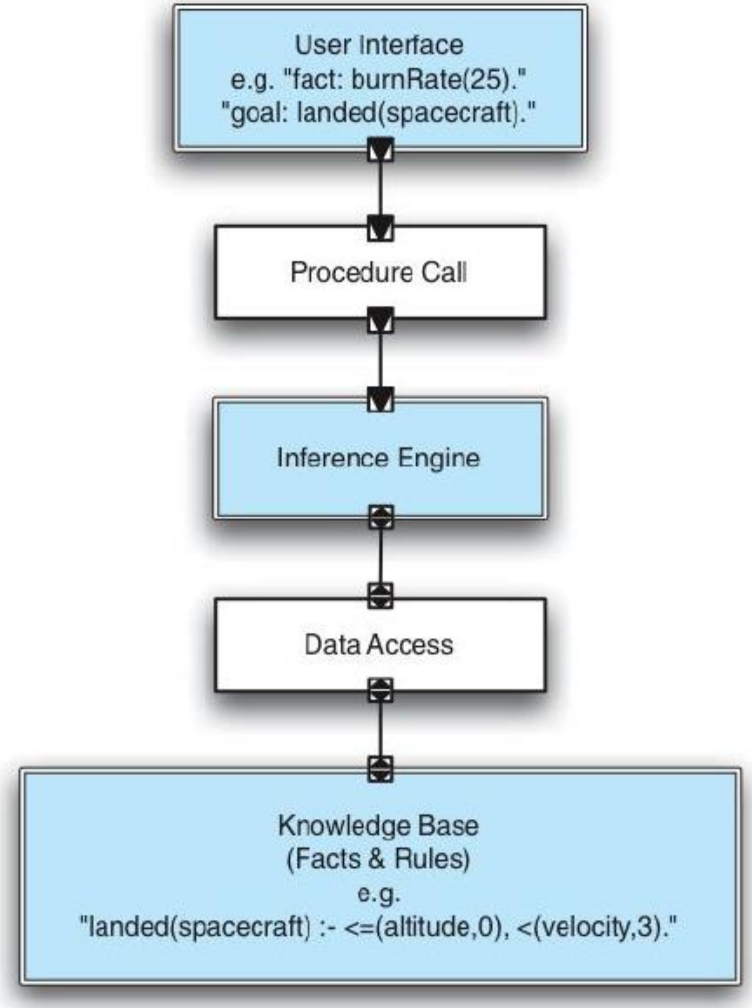
- Rule-based
  - » *Inference engine* parses user input and determines whether it is a fact/rule or a query.
  - » If it is a fact/rule, it adds this entry to the knowledge base.
  - » Otherwise, it queries the knowledge base for applicable rules and attempts to resolve the query.

Architectural Styles – Shared Memory

- Rule-based
  - **Components**: User interface, inference engine, knowledge base
  - **Connectors**: Components are tightly interconnected, with direct procedure calls and/or shared memory.
  - **Data Elements**: Facts and queries
  - **Topology**: Tightly coupled three-tier (direct connection of user interface, inference engine, and knowledge base).
  - **Quality Yield**: Behavior of the application can be very easily modified through addition or deletion of rules from the knowledge base.
  - **Typical Use:** When the problem can be understood as matter of repeatedly resolving a set of predicates.
  - **Programming Languages:** Prolog is a common language for building rule-based systems.

Architectural Styles – Shared Memory

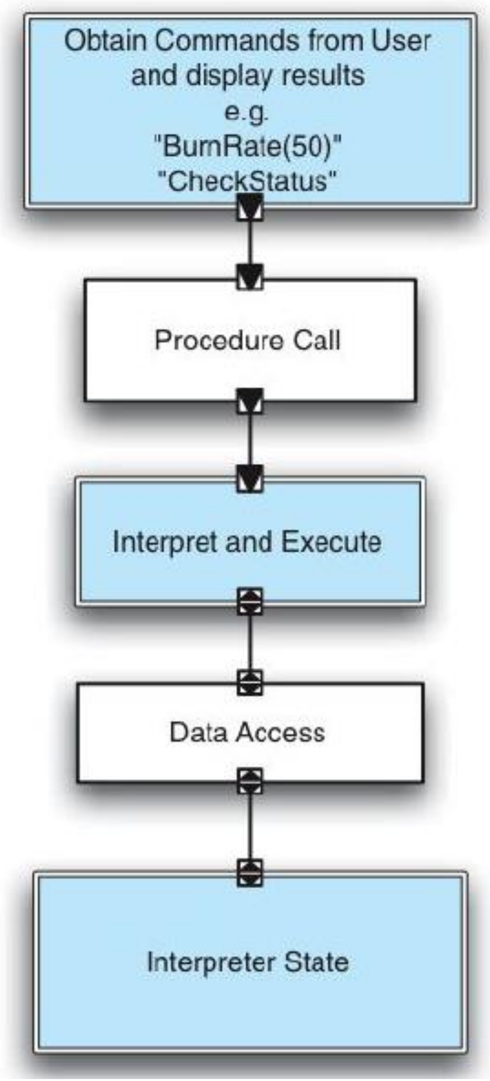
- Rule-based


User Interface
e.g. "fact: burnRate(25)."
"goal: landed(spacecraft)."
Procedure Call
Inference Engine
Data Access
Knowledge Base
(Facts & Rules)
e.g.
"landed(spacecraft) :- <=(altitude,0), <(velocity,3)."

Architectural Styles – Interpreter

- Interpreter parses and executes input commands, updating the state maintained by the interpreter
- **Components**: Command interpreter, program/interpreter state, user interface.
- **Connectors**: Typically very closely bound with direct procedure calls and shared state.
- **Data Elements**: Commands.
- **Topology**: Tightly coupled three-tier.
- **Typical Use**: Superb for end-user programmability; supports dynamically changing set of capabilities.
- **Relations to Programming Languages**: Lisp and Scheme are interpretive languages; World/Excel macros.

Architectural Styles – Interpreter



Architectural Styles – Interpreter

- Mobile-Code Style
- A data element (some representation of a program) is dynamically transformed into a data processing component.
- **Components**: "Execution dock", which handles receipt of code and state; code compiler/interpreter
- **Connectors**: Network protocols and elements for packaging code and data for transmission.
- **Data Elements**: Representations of code as data; program state; data
- **Topology**: Network.
  - » Variants: Code-on-demand, remote evaluation, and mobile agent.
- **Typical Use**:
  - » When processing large data sets in distributed locations, it becomes more efficient to have the code move to the location of these large data sets;
  - » When it is desirous to dynamically customize a local processing node through inclusion of external code.
- **Relations to programming languages**: Scripting languages (e.g. JavaScript, VBScript, ActiveX controls, Grid computing)

Architectural Styles – Interpreter

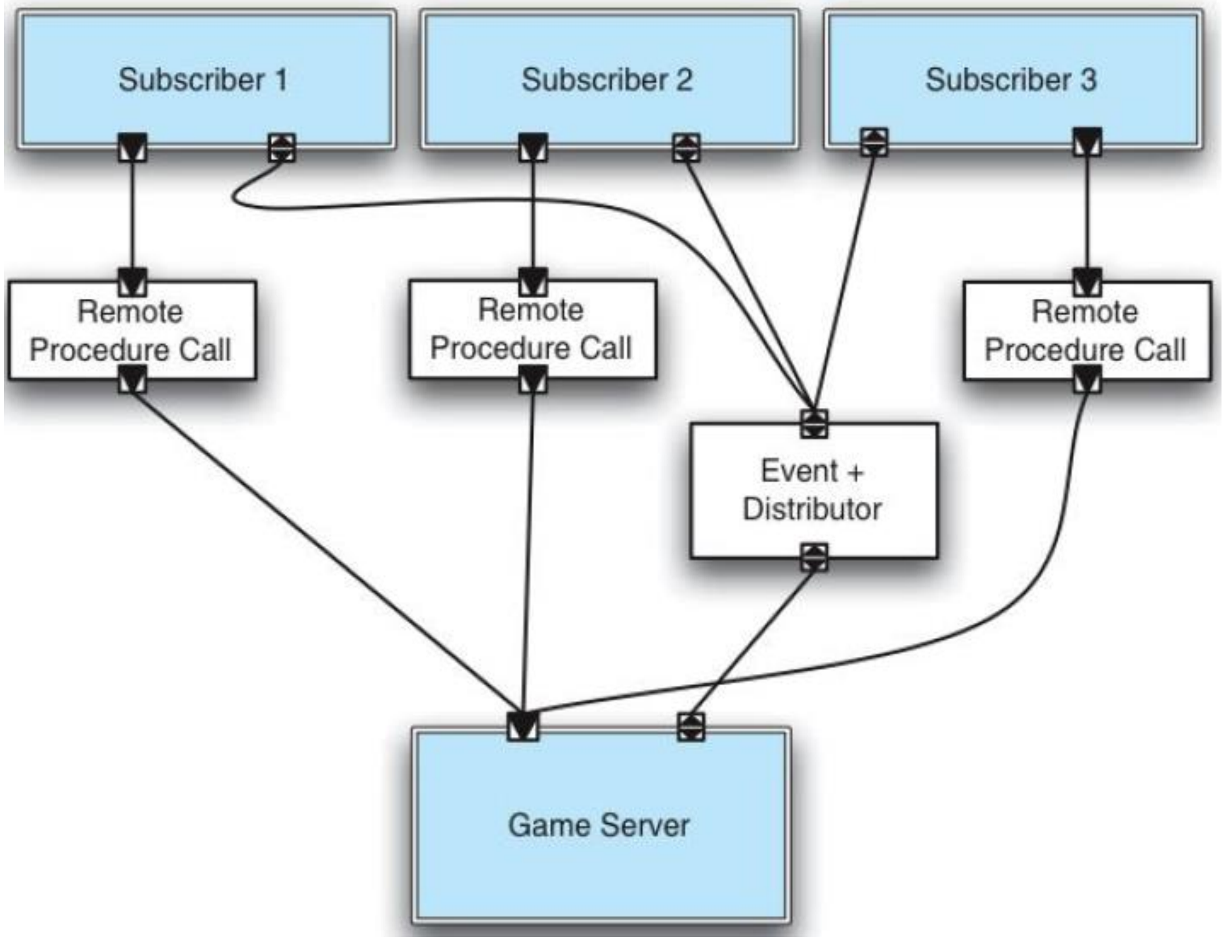
- Mobile code can be used to support three different paradigms.
  - » Code on demand
  - » Remote evaluation
  - » Mobile agent

Architectural Styles – Implicit Invocation

- Unlike the previously discussed styles, the ‘implicit invocation’ style are invoked *indirectly or directly* as a response to a *notification or an event*.
- Publisher-Subscribe
  - » Subscribers register/deregister to receive specific messages or specific content.
    - Publishers broadcast messages to subscribers either synchronously or asynchronously.
  - » **Components**: Publishers, subscribers, proxies for managing distribution
  - » **Connectors**: Typically a network protocol is required. Content-based subscription requires sophisticated connectors.
  - » **Data Elements**: Subscriptions, notifications, published information
  - » **Topology**: Subscribers connect to publishers either directly or may receive notifications via a network protocol from intermediaries
  - » **Qualities Yielded**: Highly efficient one-way dissemination of information with very low-coupling of components.

Architectural Styles – Implicit Invocation

- Publisher-Subscribe

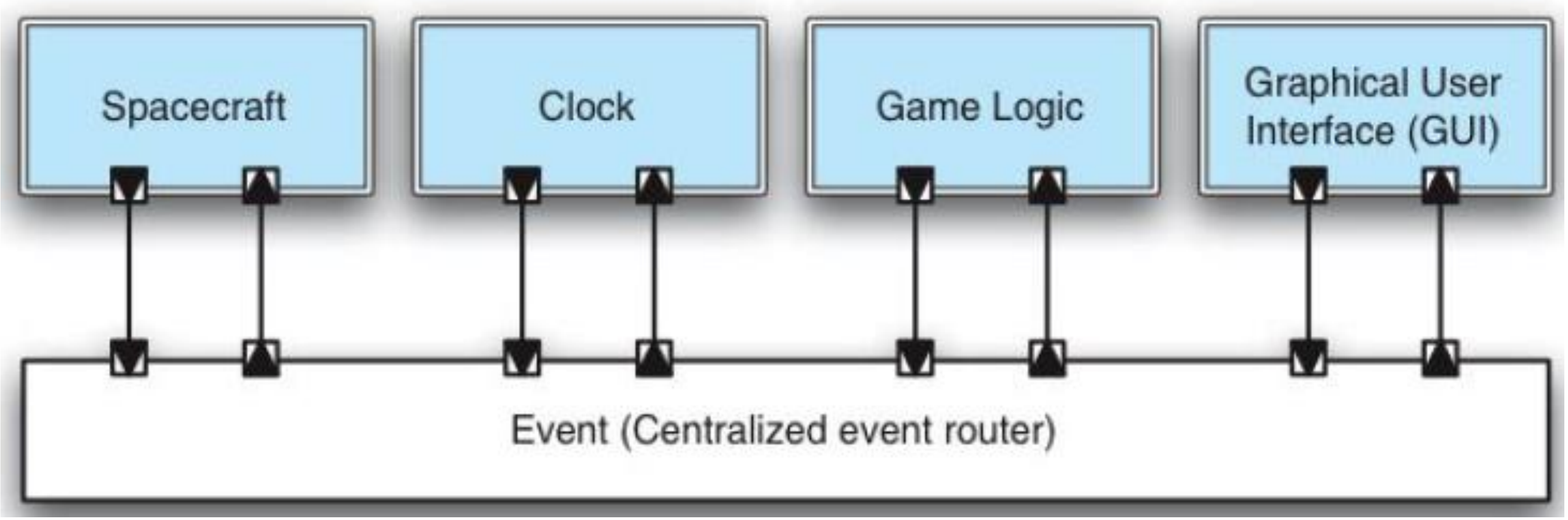


Architectural Styles – Implicit Invocation

- Event-based
  - Independent components *asynchronously* emit and receive events communicated over event buses
  - **Components**: Independent, concurrent event generators and/or consumers
  - **Connectors**: Event buses (at least one)
  - **Data Elements**: Events – data sent over the event bus
  - **Topology**: Components communicate with the event buses, not directly to each other.
  - **Variants**: Component communication with the event bus may either be push or pull based.
  - Highly scalable, easy to evolve, effective for highly distributed applications.

Architectural Styles – Implicit Invocation

- Event-based


Spacecraft
Clock
Game Logic
Graphical User Interface (GUI)
Event (Centralized event router)

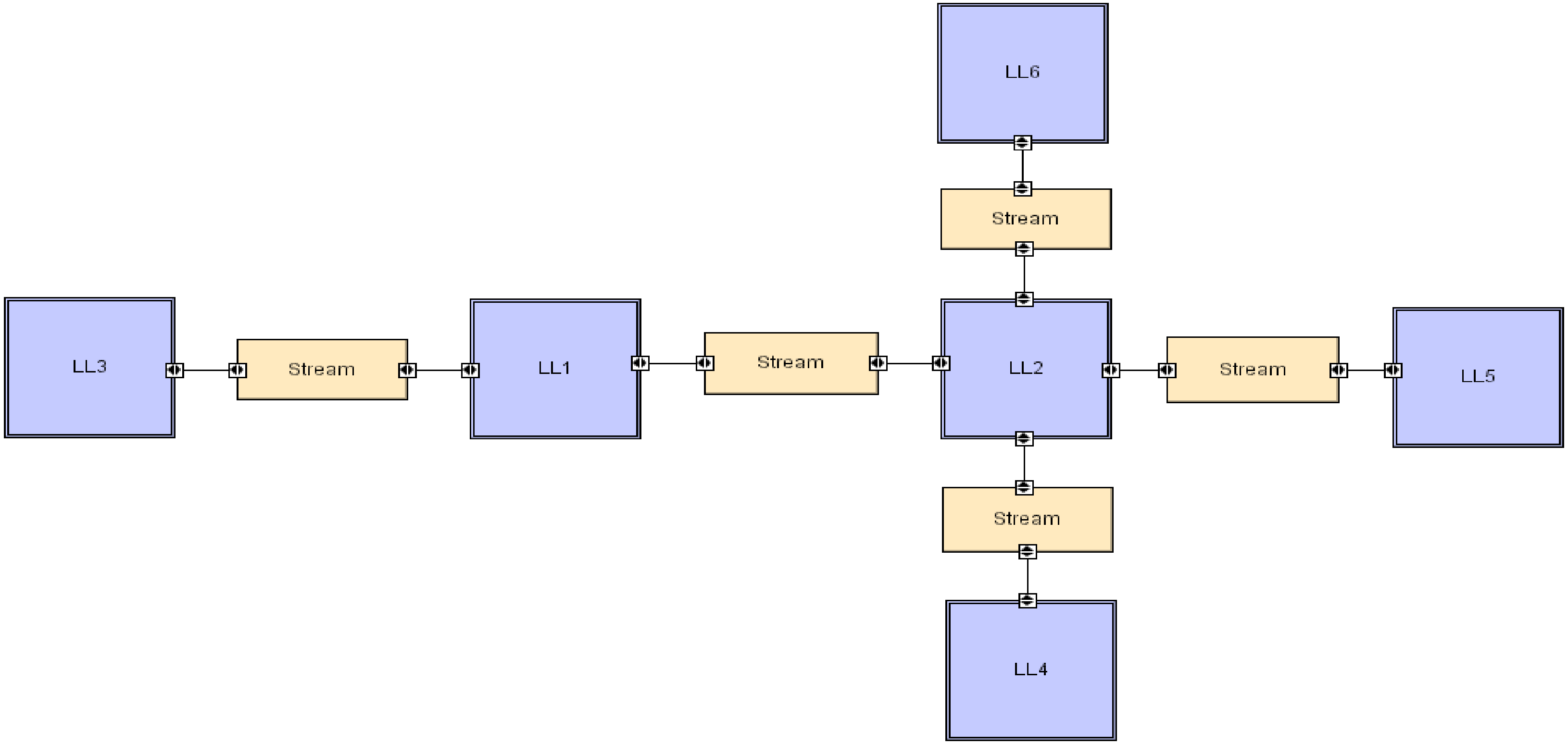
Architectural Styles – Peer-to-Peer

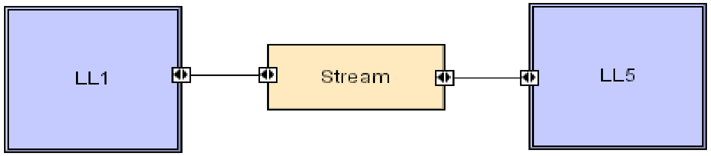
- State and behavior are distributed among peers which can act as either clients or servers.

- Peers: independent components, having their own state and control thread.
- **Connectors**: Network protocols.
- **Data Elements**: Network messages
- **Topology**: Network (may have redundant connections between peers); can vary arbitrarily and dynamically

- Supports decentralized computing with flow of control and resources distributed among peers.
  - Highly robust in the face of failure of any given node.
  - Scalable in terms of access to resources and computing power.

Architectural Styles – Peer-to-Peer



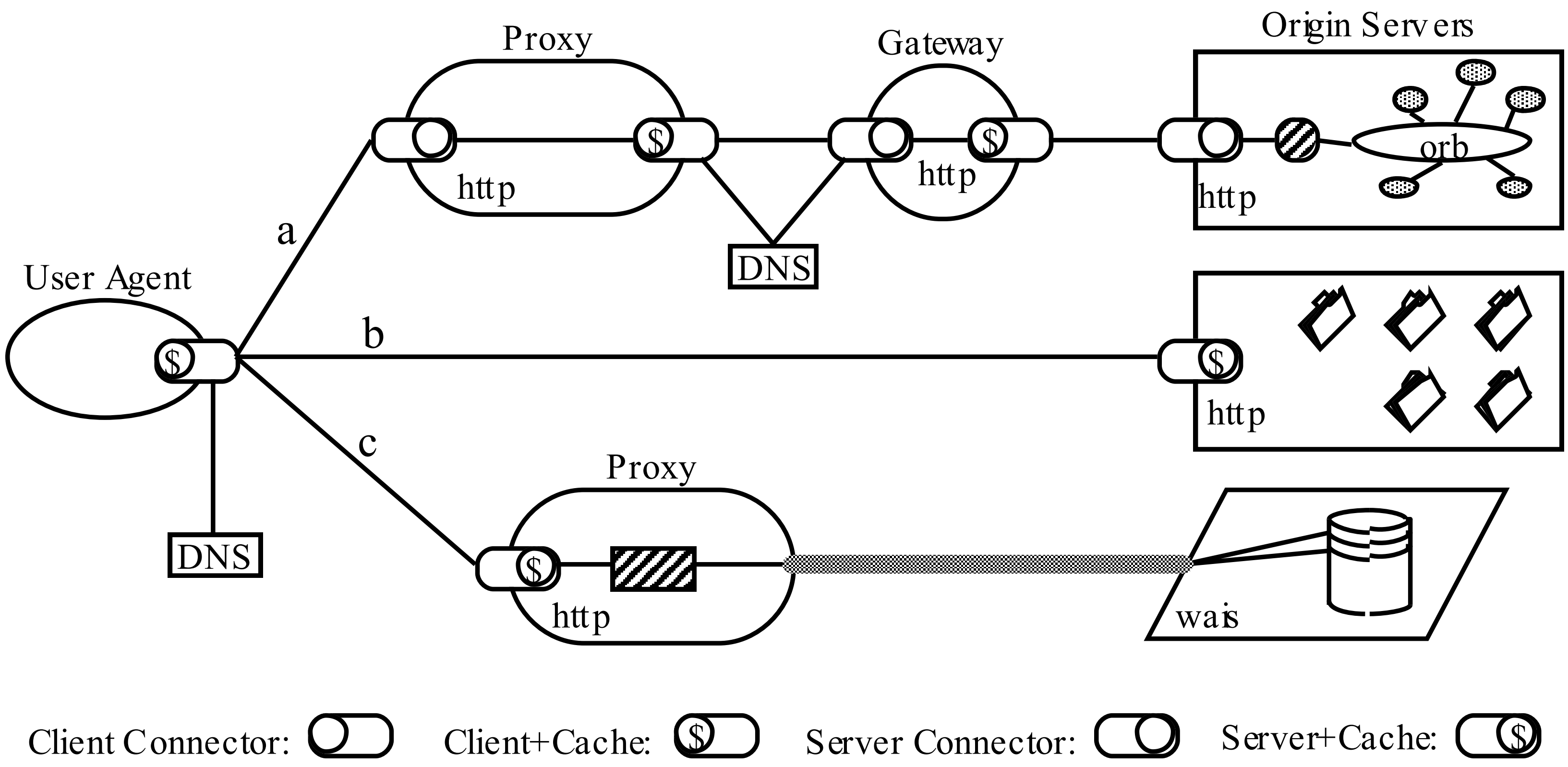


Architectural Styles – REST

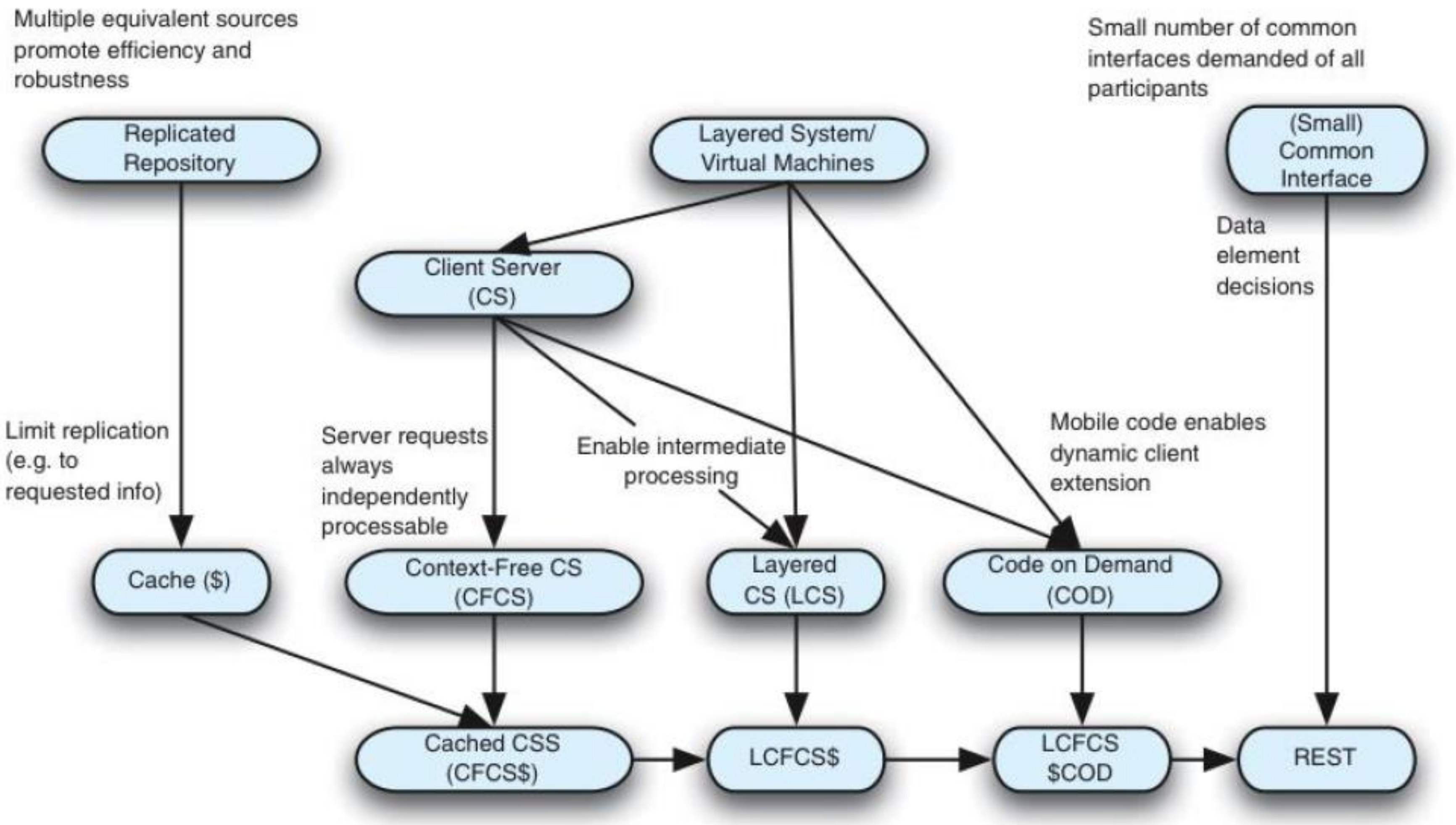
- Representation State Transfer Style (REST) Principles
  - [RP1] The key *abstraction* of information is a resource, named by an URL. Any information that can be named can be a resource.
  - [RP2] The representation of a resource is a sequence of bytes, plus representation metadata to describe those bytes. The particular form of the representation can be negotiated between REST components.
  - [RP3] All interactions are *context-free*: each interaction contains all of the information necessary to understand the request, independent of any requests that may have preceded it.

Architectural Styles – REST

- An instance of REST



Architectural Styles – REST


Multiple equivalent sources promote efficiency and robustness
Replicated Repository
Layered System/ Virtual Machines
Small number of common interfaces demanded of all participants
(Small) Common Interface
Client Server (CS)
Data element decisions
Limit replication (e.g. to requested info)
Server requests always independently processable
Enable intermediate processing
Mobile code enables dynamic client extension
Cache ($)
Context-Free CS (CFCS)
Layered CS (LCS)
Code on Demand (COD)
Cached CSS (CFCS$)
LCFCS$
LCFCS $COD
REST

Architectural Styles – REST-Components

- User agent
  - » e.g., browser
- Origin server
  - » e.g., Apache Server, Microsoft IIS
- Proxy
  - » Selected by client
- Gateway
  - » Squid, CGI, Reverse proxy
  - » Controlled by server

Architectural Styles – REST-Connectors

- Modern Web Examples
  - » client libwww, libwww-perl
  - » server libwww, Apache API, NSAPI
  - » cache browser cache, Akamai cache network
  - » resolver bind (DNS lookup library)
  - » tunnel SOCKS, SSL after HTTP CONNECT